# Desenvolupament web en entorn servidor

Àlex Salinas Tejedor

# Índex

## Introducció

En l'actualitat, la majoria d'aplicacions que s'utilitzen en entorns empresarials estan construïdes amb una arquitectura client-servidor en la qual un o diversos computadors (generalment d'una potència considerable) són servidors que proporcionen serveis a un nombre molt més gran de clients connectats a través de la xarxa. Els clients solen ser PC de propòsit general, menys potents i més orientats a l'usuari final. De vegades, els servidors són intermediaris entre els clients i altres servidors més especialitzats. Un exemple són els grans servidors de bases de dades corporatius basats en *mainframes* i/o sistemes Unix.

L'arquitectura client-servidor ha adquirit una major rellevància, ja que és el principi bàsic de funcionament de la World Wide Web: un usuari que mitjançant un navegador (client) sol·licita un servei (pàgines HTML, etc.) a un ordinador que fa les vegades de servidor. En la seva concepció més tradicional, els servidors HTTP es limitaven a enviar una pàgina HTML quan l'usuari la requeria directament o feia clic sobre un enllaç. La interactivitat d'aquest procés era mínima, ja que l'usuari podia demanar fitxers, però no enviar les seves dades personals de manera que fossin emmagatzemades en el servidor o obtingués una resposta personalitzada.

En l'apartat "*Servlets*" s'explica quines funcionalitats ens proporciona Java per tractar les peticions dels clients i es defineix què és un *servlet* i quina funcionalitat ens aporta quan codifiquem programes. La tecnologia *servlet* proporciona els mateixos avantatges del llenguatge Java quant a portabilitat i seguretat, ja que un *servlet* és una classe de Java igual que qualsevol altra, i per tant té, en aquest sentit, totes les característiques del llenguatge.

L'apartat "Formularis amb *servlets* i EJPB" té el punt de partida en els formularis HTML. Aquests formularis permeten invertir el sentit del flux de la informació. Emplenant alguns camps amb caixes de text i botons d'opció i de selecció, l'usuari pot definir les seves preferències o enviar les seves dades al servidor. Els *servlets* seran els encarregats de recollir aquesta informació, tractar-la i enviar una resposta a l'usuari. En aquest apartat s'introdueix l'arquitectura EJB, que pot ajudar els *servlets* a tractar la informació enviada per l'usuari d'una manera més còmoda per al programador, ja que porta moltes funcionalitats útils que es poden utilitzar per comprovar la validesa de les dades subministrades.

A l'apartat "Manteniment d'estat, autenticació i autorització amb *servlets* i EJB" s'expliquen diverses tècniques per emmagatzemar dades d'usuaris en el navegador client o en el servidor que es podran consultar en les properes connexions que l'usuari estableixi amb l'aplicació. D'aquesta manera es pot tenir identificat un client durant un temps determinat. Això és molt important si es vol disposar d'aplicacions que impliquin l'execució de diversos *servlets* o l'execució repetida d'un mateix *servlet*. Un clar exemple d'aplicació d'aquesta tècnica és el dels comerços via Internet, que permeten portar un carret de la compra en el qual es

van guardant aquells productes sol·licitats pel client. Aquest pot anar navegant per les diferents seccions del comerç virtual, és a dir, fent diferents connexions HTTP i executant diversos *servlets*, i malgrat això no es perd la informació continguda en el carret de la compra i se sap en tot moment que és un mateix client qui està fent aquestes connexions diferents.

La unitat descriu, des d'un vessant pràctic i teòric, aspectes essencials de la comunicació client-servidor. Tots els apartats d'aquesta unitat s'han elaborat proposant un exemple pràctic per introduir tots els conceptes abans esmentats. Es recomana que l'estudiant faci els exemples mentre els va llegint, així anirà aprenent mentre va practicant els conceptes exposats. Finalment, per treballar completament els continguts d'aquesta unitat és convenient anar fent les activitats i els exercicis d'autoavaluació.

## Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

**1.** Selecciona les arquitectures i tecnologies de programació web en entorn servidor, analitzant les seves capacitats i característiques pròpies.

- Caracteritza i diferencia els models d'execució de codi al servidor i al client web.
- Reconeix els avantatges que proporciona la generació dinàmica de pàgines web i les seves diferències amb la inclusió de sentències de guions a l'interior de les pàgines.
- Identifica els mecanismes d'execució de codi en els servidors web.
- Reconeix les funcionalitats que aporten els servidors d'aplicacions i la seva integració amb els servidors web.
- Identifica i caracteritza els principals llenguatges i tecnologies relacionats amb la programació web en entorn servidor.
- Verifica els mecanismes d'integració dels llenguatges de marques amb els llenguatges de programació en entorn servidor.
- Reconeix i avalua les eines de programació en entorn servidor.

**2.** Escriu sentències executables per un servidor web reconeixent i aplicant procediments d'integració del codi en llenguatges de marques.

- Identifica els mecanismes de generació de pàgines web a partir de llenguatges de marques amb codi encastat.
- Identifica les principals tecnologies associades.
- Utilitza etiquetes per a la inclusió de codi en el llenguatge de marques.
- Identifica la sintaxi del llenguatge de programació que s'ha d'utilitzar.
- Descriu sentències simples i comprova els seus efectes en el document resultant.
- Utilitza directives per modificar el comportament predeterminat.
- Empra els diferents tipus de variables i operadors disponibles en el llenguatge.
- Identifica els àmbits d'utilització de les variables.

**3.** Escriu blocs de sentències embeguts en llenguatges de marques, seleccionant i utilitzant les estructures de programació.

- Utilitza mecanismes de decisió en la creació de blocs de sentències.
- Fa servir i verifica el seu funcionament.
- Empra *arrays* per emmagatzemar i recuperar conjunts de dades.
- Crea i utilitza funcions.
- Usa formularis web per interactuar amb l'usuari del navegador web.
- Empra mètodes per recuperar la informació introduïda en el formulari.
- Afegeix comentaris al codi.

**4.** Desenvolupa aplicacions web embegudes en llenguatges de marques analitzant i incorporant funcionalitats segons especificacions.

- Identifica els mecanismes disponibles per al manteniment de la informació que fa a un client web concret i assenyala els seus avantatges.
- Empra sessions per mantenir l'estat de les aplicacions web.
- Utilitza galetes per emmagatzemar informació en el client web i per recuperar el seu contingut.
- Identifica i caracteritza els mecanismes disponibles per a l'autenticació d'usuaris.
- Escriu aplicacions que integrin mecanismes d'autenticació d'usuaris.
- Fa adaptacions a aplicacions web existents com a gestors de continguts o altres.
- Utilitza eines i entorns per facilitar la programació, la prova i la depuració del codi.

# 1. 'Servlets'

Els *servlets* són programes petits que s'executen dintre dels servidors d'aplicacions. En concret, en aquest apartat s'explicarà:

- Introducció als *servlets*
- Configuració dels *servlets* utilitzant anotacions i fitxers XML
- Diferents classes Java relacionades amb els *servlets*, com el `ServletContext` o contenidor d'aplicacions

En aquest apartat també parlarem del cicle de vida d'un *servlet*, els seus paràmetres inicials i com redirigir una petició.

Tots els conceptes que es veuran s'explicaran sempre partint de l'exemple. Quan acabeu aquest apartat estareu preparats per crear un *servlet*, analitzar les peticions i crear les respostes adequades.

## 1.1 'Servlet' Hola, Món

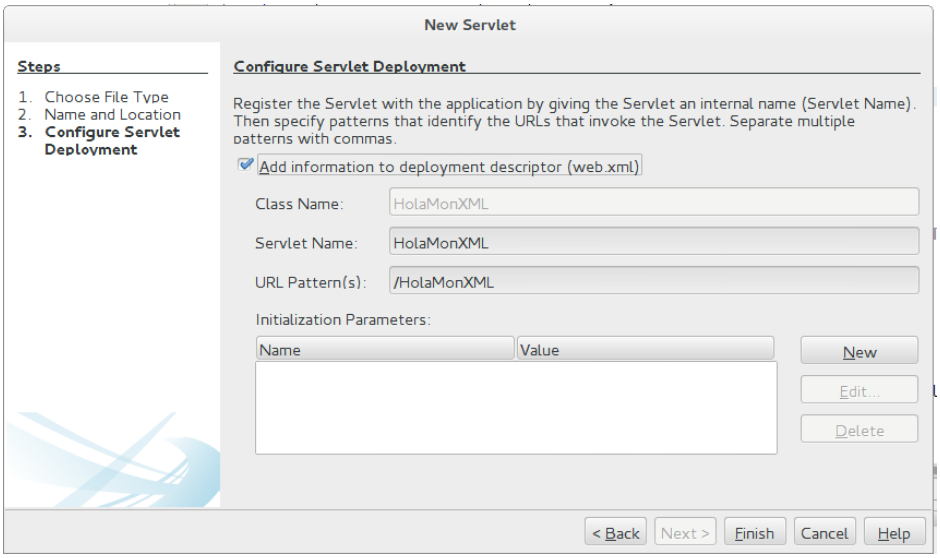
En aquest apartat aprendreu a utilitzar ***servlets***, les seves etiquetes XML i les anotacions, i s'explicarà el `ServletContext` com a contenidor d'aplicacions.

**Orígens dels 'servlets'**

Sun Microsystems va escriure la primera especificació dels servlets. Va finalitzar la versió 1.0 el juny de 1997. A partir de la versió 2.3, l'especificació dels *servlets* va ser desenvolupada subjecta al Java Community Process.

Començareu creant un nou projecte web amb Maven (vegeu la figura 3.1) i l'anomenem *servlets* (*File / New Project / Maven / Web Application*).

**FIGURA 1.1.** Procés de creació d'un 'servlet'



Una vegada s'ha creat, crearem dos *servlets* (*File / New File / Web / Servlet* ) en el paquet que indica el mateix NetBeans, en aquest cas, a *cat.ioc.m7.servlets*. De totes les opcions que podeu configurar només posareu el nom d'Hola, Món com a nom del *servlet*, les altres opcions tindran el valor per defecte. Creareu un segon *servlet* i li posareu el nom d'HolaMónXML i activareu l'opció *add information to deployment descriptor (web.xml)*. Per defecte, el codi resultant en crear el *servlet* Hola, Món és el següent:

```
1  import java.io.IOException;
2  import java.io.PrintWriter;
3  import javax.servlet.ServletException;
4  import javax.servlet.annotation.WebServlet;
5  import javax.servlet.http.HttpServlet;
6  import javax.servlet.http.HttpServletRequest;
7  import javax.servlet.http.HttpServletResponse;
8
9  /**
10  *
11  * @author ioc
12  */
13 @WebServlet(urlPatterns = {"/HolaMon"})
14 public class HolaMon extends HttpServlet {
15
16     /**
17      * Processes requests for both HTTP GET and POST
18      * methods.
19      *
20      * @param request servlet request
21      * @param response servlet response
22      * @throws ServletException if a servlet-specific error occurs
23      * @throws IOException if an I/O error occurs
24      */
25     protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
26             throws ServletException, IOException {
27         response.setContentType("text/html;charset=UTF-8");
28         try (PrintWriter out = response.getWriter()) {
29             /* TODO output your page here. You may use following sample code.
                   */
30             out.println("<!DOCTYPE html>");
31             out.println("<html>");
32             out.println("<head>");
33             out.println("<title>Servlet HolaMon</title>");
34             out.println("</head>");
35             out.println("<body>");
36             out.println("<h1>Servlet HolaMon at " + request.getContextPath() +
                   "</h1>");
37             out.println("</body>");
38             out.println("</html>");
39         }
40     }
41
42     // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click
           on the + sign on the left to edit the code.">
43     /**
44      * Handles the HTTP GET method.
45      *
46      * @param request servlet request
47      * @param response servlet response
48      * @throws ServletException if a servlet-specific error occurs
49      * @throws IOException if an I/O error occurs
50      */
51     @Override
52     protected void doGet(HttpServletRequest request, HttpServletResponse
           response)
53             throws ServletException, IOException {
54         processRequest(request, response);
```

```
55      }
56
57      /**
58       * Handles the HTTP POST method.
59       *
60       * @param request servlet request
61       * @param response servlet response
62       * @throws ServletException if a servlet-specific error occurs
63       * @throws IOException if an I/O error occurs
64       */
65      @Override
66      protected void doPost(HttpServletRequest request, HttpServletResponse
            response)
67              throws ServletException, IOException {
68          processRequest(request, response);
69      }
70
71      /**
72       * Returns a short description of the servlet.
73       *
74       * @return a String containing servlet description
75       */
76      @Override
77      public String getServletInfo() {
78          return "Short description";
79      }// </editor-fold>
80
81  }
```

El codi resultant en crear el *servlet* Hola, Món amb sintaxi XML és el següent:

```
1   import java.io.IOException;
2   import java.io.PrintWriter;
3   import javax.servlet.ServletException;
4   import javax.servlet.http.HttpServlet;
5   import javax.servlet.http.HttpServletRequest;
6   import javax.servlet.http.HttpServletResponse;
7
8   /**
9    *
10   * @author ioc
11   */
12  public class HolaMonXML extends HttpServlet {
13
14      /**
15       * Processes requests for both HTTP GET and POST
16       * methods.
17       *
18       * @param request servlet request
19       * @param response servlet response
20       * @throws ServletException if a servlet-specific error occurs
21       * @throws IOException if an I/O error occurs
22       */
23      protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
24              throws ServletException, IOException {
25          response.setContentType("text/html;charset=UTF-8");
26          try (PrintWriter out = response.getWriter()) {
27              /* TODO output your page here. You may use following sample code.
                    */
28              out.println("<!DOCTYPE html>");
29              out.println("<html>");
30              out.println("<head>");
31              out.println("<title>Servlet HolaMonXML</title>");
32              out.println("</head>");
33              out.println("<body>");
34              out.println("<h1>Servlet HolaMonXML at " + request.getContextPath()
                        + "</h1>");
35              out.println("</body>");
```

```
            out.println("</html>");
        }
    }

    // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click
        on the + sign on the left to edit the code.">
    /**
     * Handles the HTTP GET method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Handles the HTTP POST method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
        response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Returns a short description of the servlet.
     *
     * @return a String containing servlet description
     */
    @Override
    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>

}
```

Segons el codi que heu vist, intenteu contestar a la següent pregunta: quin URL s'ha d'escriure al navegador per accedir als *servlets*?

Si executeu els dos *servlets* veieu que no hi ha cap diferència, funcionen exactament igual, però si mireu el codi veieu que no és així.

La primera i fonamental diferència és la llibreria *import javax.servlet.annotation.WebServlet;*, que no hi és en el *servlet* creat amb XML perquè la seva configuració es farà en el fitxer web.xml. En canvi, al *servlet* creat per defecte, la seva configuració es posarà com a anotacions dintre del mateix *servlet*, que es llegiran en el moment de la seva compilació.

Però abans de començar veient les diferències contesteu a la següent pregunta: què és un *servlet* i com funciona?

Un ***servlet*** és un programa del costat del servidor que s'utilitza per generar pàgines web dinàmiques. Genera pàgines web com a resposta d'una petició rebuda des del client (navegador).

El funcionament d'un *servlet* és el mateix tant amb XML o amb Annotations (anotacions). El *servlet* implementa el costat servidor de la comunicació client-servidor.

Un *servlet* pot rebre la petició de dues maneres diferents: amb el mètode `GET` o amb el mètode `POST` d'HTTP. Amb el mètode `GET`, l'habitual, un navegador accedeix a una pàgina web. El mètode `POST` és el mètode utilitzat en l'enviament de dades al servidor des d'un formulari web.

Un *servlet* és capaç de rebre una invocació i generar una resposta, com, per exemple, enviar una pàgina web quan algú accedeix al *servlet* mitjançant la seva URL.

Si utilitzeu el mètode `GET` s'executarà la funció `doGet`, i si empreu NetBeans crearà una funció com aquesta:

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

Igualment, si s'utilitza el mètode `POST` s'executarà la funció `doPost`:

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

Com veieu, les dues funcions anteriors fan la crida a la mateixa funció protegida, anomenada `processRequest`. Això és perquè NetBeans suposa que volem el mateix comportament tant si es fa una petició amb el mètode `GET` com amb el mètode `POST`.

La funció `processRequest` ha de generar una resposta segons els paràmetres de la petició. En aquests cas, en ser un programa molt senzill, només volem veure per pantalla una pàgina web amb la benvinguda. La pàgina web que veieu la creeu de manera dinàmica dintre de la funció. Vegeu com s'implementa la resposta:

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
        /* TODO output your page here. You may use following sample code. */
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet HolaMon</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet HolaMon at " + request.getContextPath() + "</
            h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Com veieu, esteu escrivint el codi HTML directament a l'objecte resposta (`response.getWriter()`). Escriviu, utilitzant l'objecte `PrinterWriter`, la pàgina Hola, Món que volem que vegi l'usuari.

El funcionament intern dels dos *servlets* és el mateix.

La configuració del *servlet* amb XML es fa mitjançant el fitxer **web.xml**, conegut com a descriptor de desplegament (*deployment descriptor*).

El servidor Apache Tomcat és un exemple de contenidor web, no comercial, que suporta l'execució de *servlets* definits mitjançant el fitxer web.xml.

El fitxer **web.xml** es troba dintre de la carpeta *WEB-INF* i conté la informació corresponent al nom i a l'URL dels *servlets*. Però bàsicament, aquest fitxer informa el Servlet Container de la classe que ha d'executar (*servlet*) per a un URL donat.

El fitxer web.xml conté la següent informació:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="
       http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
       xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1
       .xsd">
3      <servlet>
4          <servlet-name>HolaMonXML</servlet-name>
5          <servlet-class>HolaMonXML</servlet-class>
6      </servlet>
7      <servlet-mapping>
8          <servlet-name>HolaMonXML</servlet-name>
9          <url-pattern>/HolaMonXML</url-pattern>
10     </servlet-mapping>
11     <session-config>
12         <session-timeout>
13             30
14         </session-timeout>
15     </session-config>
16 </web-app>
```

En el fitxer anterior observeu com es defineix un *servlet* i quin URL s'ha d'utilitzar per poder emprar-lo des del navegador. L'etiqueta `servlet-name` conté el nom del *servlet* i l'etiqueta `servlet-class`, la classe on està codificat.

Una vegada s'ha definit el *servlet* s'ha d'informar de l'URL que s'utilitzarà per executar el *servlet* anterior. Aquesta informació es defineix dintre de l'etiqueta `servlet-mapping`. En concret, s'ha de dir el nom del *servlet* i l'URL associat. Les etiquetes XML són *servlet-name* per al nom del *servlet* i `url-pattern` per definir l'URL que s'utilitzarà.

Tota aquesta informació es pot reduir molt utilitzant Annotations (anotacions). Vegeu la informació que es genera en el *servlet* per defecte per definir tota la informació anterior:

```
1  @WebServlet(urlPatterns = {"/HolaMon"})
```

L'anotació `@WebServlet` s'utilitza per declarar una classe de tipus *servlet*. Aquesta classe ha d'heretar de la classe `HttpServlet` i s'empra per configurar un mapatge URL. Vegeu-ne un altre exemple:

```java
@WebServlet(
    name = "ServletExemple",
    description = "Un exemple d'anotació Servlet",
    urlPatterns = {"/ServletExemple"}
)
public class ServletExemple extends HttpServlet {
    // codi...
}

//o amb més d'una URL
@WebServlet(
    urlPatterns = {"/foo", "/bar", "/cool"}
)
public class ServletExemple extends HttpServlet {
    // codi...
}
```

Vegeu que és molt més ràpid utilitzar les anotacions que el marcatge XML per definir la configuració dels *servlets*.

Ara ja heu de poder contestar a aquesta pregunta: quin URL s'ha d'escriure per accedir als *servlets*?

L'URL que s'ha d'escriure seria semblant a: localhost:8080/servlets/HolaMon.

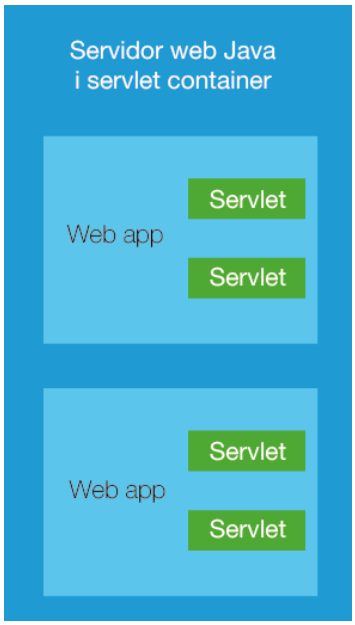
En general:

```
http://ip_servidor:port/nom_aplicacio/SERVLET_URL_PATTERN
```

En el vostre cas, el servidor Glassfish us dóna el port 8080 per poder connectar-nos, i la IP és el mateix PC. El nom de l'aplicació correspon al nom del projecte que heu creat amb Netbeans, i el `servlet-URL-pattern` correspon a l'URL definit, amb l'anotació `@WebServlet` o bé amb l'etiqueta `url-pattern` del fitxer web.xml.

Com sabeu, un servidor pot tenir més d'una aplicació funcionant al mateix temps. Dintre de cada aplicació hi pot haver més d'un *servlet* actiu, atès que totes les aplicacions estan dintre del Servlet Container (vegeu la figura 3.2).

**FIGURA 1.2.** Java Servlet Container



Tots els *servlets* definits en una mateixa aplicació web comparteixen recursos i la informació web de l'aplicació. Per poder accedir a tota aquesta informació s'ha d'accedir al `ServletContext`, que dóna un conjunt de mètodes als *servlets* perquè es puguin comunicar.

> El `ServletContext` és un objecte que conté **metainformació sobre l'aplicació**. Els atributs emmagatzemats estan disponibles a tots els *servlets* de l'aplicació, fins i tot entre diferents peticions i sessions.

S'hi pot accedir via l'objecte `HttpRequest` de la següent manera:

```
1 ServletContext context = request.getSession().getServletContext();
```

Aquests atributs es troben emmagatzemats en memòria del Servlet Container, la qual cosa vol dir que estan disponibles a tots els clients de l'aplicació. En canvi, els atributs de sessió només estan disponibles per a l'usuari que ha creat la sessió.

Exemple de creació d'un atribut en el `ServletContext`:

```
1 context.setAttribute("atribut", "valor_del_atribut");
```

Exemple de lectura d'un atribut en el `ServletContext`:

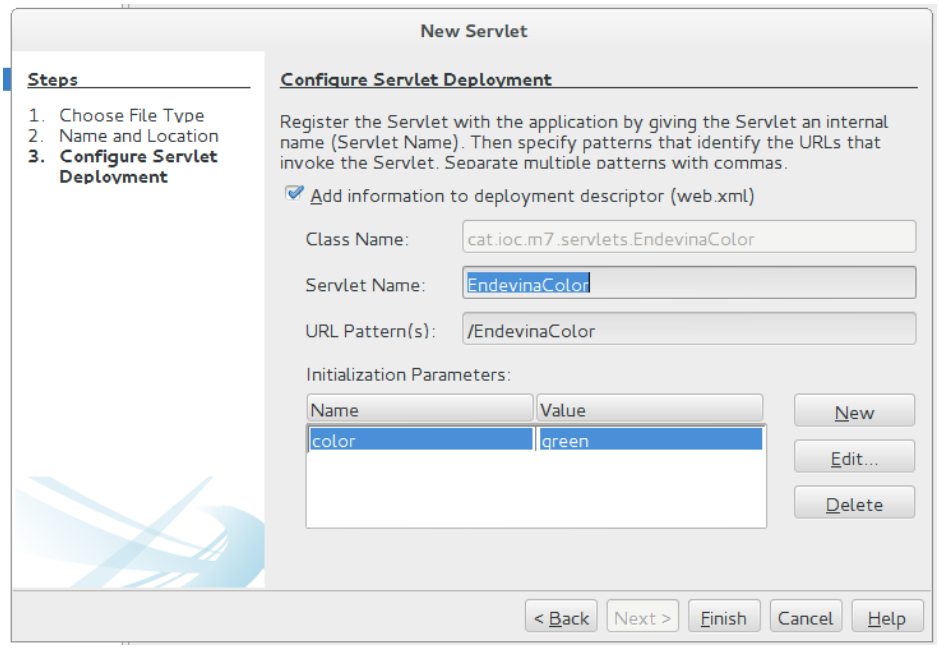
```
1 Object valorAtribut = context.getAttribute("atribut");
```

## 1.2 'Servlet' EndevinaColor

En aquest exemple voleu codificar un joc d'endevinació molt senzill. El programa tindrà configurat un color com a paràmetre inicial (constant) i li donareu a l'usuari unes quantes opcions. L'usuari haurà d'endevinar el color configurat.

Primer creareu un *servlet* nou (*File / New File / Web / Servlet*) en el mateix projecte Maven de l'exemple anterior i l'anomenareu EndevinaColor. Durant la seva creació seleccioneu que volem afegir la seva configuració en el fitxer web.xml i introduireu un paràmetre inicial nou (vegeu la figura 3.3). El paràmetre inicial introduït es diu *color*, i el seu valor correspondrà al color que l'usuari ha d'endevinar.

**FIGURA 1.3.** Configurar paràmetres d'inicialització amb l'IDE NetBeans




Fixeu-vos en el codi afegit al fitxer de configuració web.xml:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="
       http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
       xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1
       .xsd">
3  ...
4      <servlet>
5          <servlet-name>EndevinaColor</servlet-name>
6          <servlet-class>cat.ioc.m7.servlets.EndevinaColor</servlet-class>
7          <init-param>
8              <param-name>color</param-name>
9              <param-value>green</param-value>
10         </init-param>
11     </servlet>
12
13     <servlet-mapping>
14         <servlet-name>EndevinaColor</servlet-name>
15         <url-pattern>/EndevinaColor</url-pattern>
16     </servlet-mapping>
17 ...
18 </web-app>
```

Si us hi fixeu, s'especifica el nou *servlet* anomenat EndevinaColor i la seva classe Java associada.

A Java EE existeixen dos tipus de descriptors de desplegament (*deploymnet descriptors*): *Java EE deployment descriptors* i *runtime deployment descriptors*. El fitxer *web.xml* és un exemple de fitxer de desplegament estàndard per a *servlets* de Java. I el fitxer *sun-web-xml* és un exemple de fitxer de configuració específic per al servidor Glassfish.

> Les etiquetes `init-param` serveixen per definir els paràmetres inicials o constants als quals podrà accedir el *servlet* en el moment d'execució.

L'avantatge d'utilitzar paràmetres afegits a la seva configuració és que si es volgués canviar el seu valor no s'hauria de modificar ni compilar el *servlet*. Un altre paràmetre que s'ha configurat és l'URL EndevinaColor, que correspon al *servlet* EndevinaColor (*servlet-mapping*).

Per crear el joc d'endevinació es necessitaran dos fitxers. El primer serà el *servlet* EndevinaColor.java, i el segon serà la pàgina inicial que es mostrarà a l'usuari

amb les opcions que pot escollir. El nom de la pàgina serà EndevinaColor.html i es guardarà a la carpeta per defecte *Web Pages*. Aquesta pàgina serà un HTML amb un llistat d'enllaços, cadascun dels quals informarà el *servlet* del color escollit per l'usuari. Com que el paràmetre es passa junt amb l'URL del *servlet*, el mètode HTTP utilitzat per enviar les dades és `GET`. Vegeu un exemple de pàgina HTML amb el llistat d'enllaços:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Start Page</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>Endevina el color configurat:</h1>
        <a href='EndevinaColor?color=white'>blanc</a>
        <a href='EndevinaColor?color=red'>vermell</a>
        <a href='EndevinaColor?color=blue'>blau</a>
        <a href='EndevinaColor?color=yellow'>groc</a>
        <a href='EndevinaColor?color=green'>verd</a>
        <a href='EndevinaColor?color=black'>negre</a>
    </body>
</html>
```

Com podeu observar, depenent de l'enllaç que esculli l'usuari s'enviarà al *servlet* un color diferent, que correspon a la proposta que li fa l'usuari. El *servlet* haurà de comparar aquest color amb el color configurat en el fitxer web.xml. Si són iguals, l'usuari haurà endevinat el color, i si no ho són l'usuari haurà perdut i ho podrà tornar a intentar. A continuació podeu veure aquest comportament traduït a codi Java i utilitzat per implementar el *servlet* EndevinaColor:

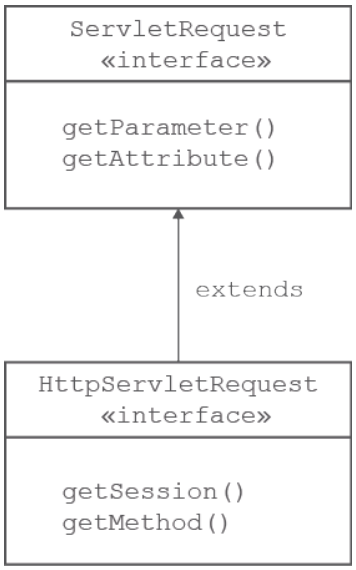
```
protected void processRequest(HttpServletRequest request, HttpServletResponse
    response)throws ServletException, IOException {
   response.setContentType("text/html;charset=UTF-8");
   try (PrintWriter out = response.getWriter()) {

      out.println("<!DOCTYPE html>");
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Endevina el color</title>");
      out.println("</head>");
      out.println("<body>");

      String endevinat = "Llàstima, has perdut!";

      //L'usuari ha seleccionat un color i ho ha enviat.
      String colorUsuari = request.getParameter("color");

      //S'ha configurat un paràmetre que conté el color a endevinar:
      String colorInicial = getServletConfig().getInitParameter("color");

      if(colorInicial.toLowerCase().equals(colorUsuari.toLowerCase())){
         endevinat = "Felicitats! Has endevinat el color.";
      }

      out.println("<h1>" + endevinat +"</h1>");
      out.println("<a href='EndevinaColor.html'>Tornar a intentar<a/>");
      out.println("</body>");
      out.println("</html>");

   }
}
```

Si voleu accedir al color que l'usuari ha enviat s'ha de mirar la seva petició (*request*). Els navegadors web envien molta informació al servidor web que aquest no pot llegir directament perquè forma part de la capçalera de la petició HTTP.

La llibreria (API) dels *servlets* defineix dues interfícies on s'encapsula la petició de l'usuari en forma d'objecte. Les classes son `javax.servlet.ServletRequest` i `javax.servlet.http.HttpServletRequest`.

Quina diferència hi ha entre aquestes dues classes? (vegeu la figura 3.4).

**FIGURA 1.4.** La classe HttpServletRequest hereta de la classe ServletRequest




La classe `HttpServletRequest` hereta la classe `ServletRequest` per afegir mètodes que es relacionen amb la capçalera del protocol HTTP. Per exemple, els mètodes `getSession` o `getCookies` són mètodes de la classe `HttpServletRequest` però no de la classe `ServletRequest`. És així perquè la classe `HttpServletRequest` ha analitzat la petició i ha creat els mètodes necessaris per accedir a la informació d'una manera més intuïtiva.

La funció `processRequest` dóna accés a la classe `HttpServletRequest`, mitjançant la qual podrem accedir al paràmetre enviat per l'usuari.

```
1 String colorUsuari = request.getParameter("color");
```

El mètode `getParameter(nom_parametre)` retorna el valor del paràmetre enviat a la petició de l'usuari. En el cas que no existeixi el paràmetre, el resultat de l'assignació de la variable `colorUsuari` seria `NULL` (`colorUsuari=NULL`).

Ara només queda accedir al paràmetre inicial del *servlet* per poder saber si l'usuari ha encertat amb el color. Per accedir al color definit en el *servlet* s'utilitza la següent instrucció:

```
1 String colorInicial = getServletConfig().getInitParameter("color");
```

L'objecte `ServletConfig` s'utilitza per obtenir informació de configuració del fitxer web.xml. Aquest objecte és creat pel contenidor web per a cada *servlet*.

Si la informació de configuració es defineix a l'arxiu web.xml no cal canviar el *servlet*. Per tant, és més fàcil d'administrar l'aplicació web si els paràmetres del *servlet* només es modifiquen de tant en tant.

L'objecte `ServletConfig` sempre està disponible per al *servlet* durant la seva execució. Una vegada que el *servlet* ha completat l'execució, l'objecte `ServletConfig` serà eliminat pel contenidor.

Llavors, l'avantatge principal d'utilitzar l'objecte `ServletConfig` és que no cal editar l'arxiu *servlet* si la informació inicial s'ha introduït a l'arxiu web.xml.

Una vegada tenim la variable inicial 'color' ja podem procedir a comparar-la amb el color que ha enviat l'usuari.

```
String endevinat = "Llàstima, has perdut!";
...
if(colorInicial.toLowerCase().equals(colorUsuari.toLowerCase())){
     endevinat = "Felicitats! Has endevinat el color.";
}

out.println("<h1>" + endevinat +"</h1>");
```

Utilitzant la variable `endevinat` mostrem la informació a l'usuari. En el cas que sigui el mateix color es mostrarà la frase "Felicitats! Has endevinat el color"; en cas contrari es mostrarà "Llàstima, has perdut!".

## 1.3 'Servlet' Publicitat als nous

En aquest exemple es vol crear un *servlet* que identifiqui si és la primera vegada que s'accedeix a la pàgina. En el cas que sigui la primera vegada es mostrarà una pàgina amb un text publicitari; si no és la primera vegada se'n mostrarà una altra.

Començareu creant un nou *servlet* (*File / New File / Web / Servlet*) anomenat Publicitat dintre del mateix projecte Maven utilitzat en els exemples anteriors. Durant la creació del *servlet* hi afegireu un paràmetre addicional anomenat *URL* que contindrà l'URL del patrocinador de l'aplicació (el valor el podeu inventar). A més a més, hi incorporareu la descripció del *servlet* al fitxer de configuració web.xml.

Una possible implementació del *servlet* demanat és el següent:

```
public class Publicitat extends HttpServlet {
      private int visites;

    private int num;
    private String urlPublicitat;
    private HashMap ip;

    @Override
    public void init (ServletConfig config) throws ServletException
    {
        super.init(config);
        this.ip = new HashMap();
        this.num++;
```

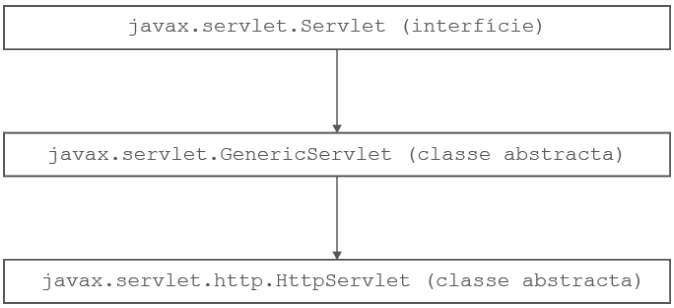
```
14          this.visites = 0;
15          this.urlPublicitat = getServletConfig().getInitParameter("url");
16      }
17
18      @Override
19      public void service(HttpServletRequest request, HttpServletResponse
            response) throws ServletException, IOException{
20       response.setContentType("text/html;charset=UTF-8");
21          try (PrintWriter out = response.getWriter()) {
22
23              out.println("<!DOCTYPE html>");
24              out.println("<html>");
25              out.println("<head>");
26              out.println("<title>Servlet Publicitat</title>");
27              out.println("</head>");
28              out.println("<body>");
29
30              String requestIp = request.getRemoteAddr();
31
32              if(this.ip.containsKey(requestIp)){
33
34                  noEsLaPrimeraVegada(out);
35              }
36              else{
37
38                  esLaPrimeraVegada(requestIp, out);
39              }
40
41              out.println("<h5>S'han fet " + this.visites +" visites</h5>");
42              out.println("<h5>S'ha cridat el mètode init " + this.num +" vegades
                    </h5>");
43              out.println("</body>");
44              out.println("</html>");
45
46              this.visites++;
47          }
48      }
49
50      private void noEsLaPrimeraVegada(PrintWriter out){
51          out.println("<h1>Gràcies per tornar a la pàgina web. Ja no veuràs el
                patrocinador.</h1>");
52      }
53
54      private void esLaPrimeraVegada(String requestIp, PrintWriter out){
55
56          this.ip.put(requestIp, "");
57
58          out.println("<h1>És la primera vegada que accedeixes a la pàgina.
                Benvingut.</h1>");
59          out.println("<p style='color:red;'>Accedeix al nostre patrocinador
                clicant al següent enllaç:</p>");
60          out.println("<a href='" + this.urlPublicitat + "'>Pàgina web del
                patrocinador</a>");
61      }
62
63  }
```

On es troben implementats els mètodes `processRequest`, `doGet()` i `doPost()`?

En aquest cas no s'ha implementat cap d'aquests mètodes. S'ha anat als orígens i s'ha intentat contestar a la següent pregunta: quins són els mètodes bàsics que ha d'implementar un *servlet*?

Per poder contestar a la pregunta heu de saber de quines classes hereta un *servlet* (vegeu la figura 3.5).

**FIGURA 1.5.** Herència d'un 'servlet'




La interfície ***servlet*** és la interfície més genèrica. Tots els mètodes d'aquesta interfície s'han d'implementar. Els mètodes són: `init`, `service`, `destroy`, `getServletInfo` i `getServletConfig`.

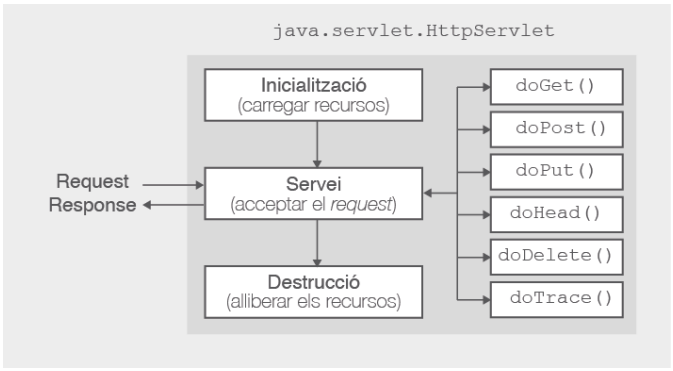
La classe abstracta `GenericServlet` implementa la interfície *servlet* donant una implementació genèrica a tots el mètodes de la interfície *servlet* excepte per al mètode `service`, que està definit com a abstracte. Qualsevol que vulgui implementar un *servlet* pot heretar d'aquesta classe i només hauria d'implementar el mètode `service`. També podria donar una implementació alternativa a qualsevol altre mètode, si el sobreescriu.

**Protocol HTTP**

El protocol HTTP està basat en uns mètodes que indiquen l'acció a realitzar sobre un recurs web determinat. Els més coneguts són els mètodes *GET* i *POST*, però n'existeixen alguns més com: *HEAD, PUT, DELETE, TRACE, OPTIONS, CONNECT i PATCH*.

Finalment, la classe abstracta `HttpServlet` hereta de la classe `GenericServlet` implementant els mètodes necessaris per donar una solució adaptada al protocol HTTP. Aquesta classe, tot i ser abstracta, no té cap mètode abstracte. El mètode `service` ha estat reemplaçat pels mètodes `doGet`, `doPost`, etc., amb els mateixos paràmetres que el mètode `service` (vegeu la figura 3.6).

**FIGURA 1.6.** Implementació del mètode service en submètodes




El codi corresponent a una implementació senzilla del mètode `service` de la classe `HttpRequest` podria ser el següent:

```
1  protected void service(HttpServletRequest req, HttpServletResponse resp) {
2      String method = req.getMethod();
3
4      if (method.equals(METHOD_GET)) {
5              doGet(req, resp);
6      } else if (method.equals(METHOD_HEAD)) {
7          doHead(req, resp);
8      } else if (method.equals(METHOD_POST)) {
9          doPost(req, resp);
10
11     } else if (method.equals(METHOD_PUT)) {
12         doPut(req, resp);
```

```
13
14      } else if (method.equals(METHOD_DELETE)) {
15          doDelete(req, resp);
16
17      } else if (method.equals(METHOD_OPTIONS)) {
18          doOptions(req,resp);
19
20      } else if (method.equals(METHOD_TRACE)) {
21          doTrace(req,resp);
22
23      } else {
24          resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
25      }
26  }
```

Com a regla general no s'ha de sobreescriure mai el mètode `service` si s'està responent a una petició feta amb el **protocol HTTP**. La solució més adient és sobreescriure els mètodes `doGet` o `doPost`.

### 1.3.1 Funcionament d'un 'servlet'. Cicle de vida

Contestant a la pregunta "Quins són els mètodes bàsics que ha d'implementar un *servlet*?", són tres:

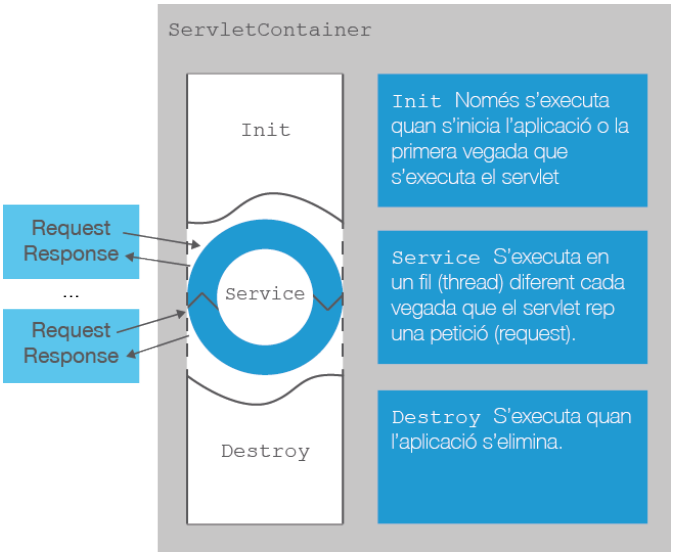
- `init`: s'executa quan s'inicia l'aplicació o la primera vegada que s'executa el *servlet*.
- `service`: s'executa en un fil (*thread*) diferent cada vegada que el *servlet* rep una petició (*request*).
- `destroy`: s'executa quan s'elimina l'aplicació.

Aquests tres mètodes constitueixen el cicle de vida d'un *servlet* (vegeu la figura 3.7).

**Threads**

Els fils d'execució (*threads*) són les unitats més petites d'execució d'un programa. Un procés està format per diversos *threads* que poden ser executats simultàniament. La majoria dels llenguatges de programació disposen de llibreries per programar-los. Les aplicacions típiques que els utilitzen son les aplicacions gràfiques i les aplicacions basades en client-servidor.

**Figura 1.7.** Cicle de vida d'un 'servlet'



La primera vegada que s'executa el *servlet* es crea l'objecte que representa aquest *servlet* i s'executa el mètode `init`. Aquest objecte, que representa el *servlet*, no s'elimina fins que no s'elimina tota l'aplicació.

Així, qualsevol petició (*request*) que rebi de qualsevol client la manipularà el mateix objecte, i qualsevol variable que tingui el *servlet* es compartirà entre peticions de diferents clients. Cada petició que rebi el *servlet* s'executarà en un fil (*thread*) diferent, i poden executar-se diferents peticions de manera concurrent (a la mateixa vegada). El mètode `service` és el mètode encarregat d'executar les peticions dels clients.

Finalment, quan l'aplicació és eliminada s'eliminen tots els *servlets* associats. En aquest moment s'executa el mètode `destroy` per alliberar els recursos que tingui, com per exemple tancar una connexió a basede dades.

### El mètode init del 'servlet' Publicitat

El mètode `init` té la peculiaritat que només s'executarà una vegada, en crear l'objecte del *servlet*. Podeu pensar en aquest mètode com si fos el seu constructor, però no ho és. El codi que hi haurà dintre d'aquest mètode serà el d'inicialització de les variables privades del *servlet*.

Fixem-nos en el mètode `init` del *servlet* Publicitat:

```
1 @Override
2 public void init (ServletConfig config) throws ServletException
3 {
4     super.init(config);
5     this.ip = new HashMap();
6     this.num++;
7     this.visites = 0;
8     this.urlPublicitat = getServletConfig().getInitParameter("url");
9 }
```

S'han inicialitzat totes les variables privades. S'utilitza un objecte del tipus *HashMap* per emmagatzemar les diferents IP del clients que es connectin al *servlet.*

S'han creat dues variables, una per comptar el nombre de vegades que s'executa la funció `init` (variable `num` ) i una altra per comptar el nombre de vegades que s'executa el mètode `service` (variable `visites`).

A més a més, s'ha recuperat el paràmetre inicial `url` que es troba en el fitxer web.xml. Aquest paràmetre correspon a l'URL del patrocinador de la pàgina web.

> El mètode `init` s'utilitza principalment per a la **inicialització** del *servlet*, és a dir, per crear o recuperar objectes que s'utilitzaran durant l'execució del mètode `service`.

I la instrucció `super.init(config);`? És imprescindible, atès que associa l'objecte `ServletConfig` al *servlet* Publicitat. Sense aquest objecte no podríem accedir als paràmetres inicials emmagatzemats en el fitxer web.xml.

### El mètode service del 'servlet' Publicitat

El mètode `service` s'executarà cada cop que un client accedeixi al *servlet*. Així, cada vegada que un navegador accedeixi a l'URL del *servlet* Publicitat s'executarà el següent codi:

```
public void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException{
     response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Publicitat</title>");
            out.println("</head>");
            out.println("<body>");

            String requestIp = request.getRemoteAddr();

            if(this.ip.containsKey(requestIp)){

                noEsLaPrimeraVegada(out);
            }
            else{

                esLaPrimeraVegada(requestIp, out);
            }

            out.println("<h5>S'han fet" + this.visites +" visites</h5>");
            out.println("<h5>S'ha cridat el mètode init " + this.num +" vegades
                </h5>");
            out.println("</body>");
            out.println("</html>");

            this.visites++;
        }
    }
```

El mètode `service` té accés, igual que els mètodes `doGet` i `doPost`, a l'objecte `request` de la petició i a l'objecte `response` per donar una resposta.

El codi que us permet decidir si és la primera vegada que l'usuari accedeix a la pàgina o no ho és és el següent:

```
String requestIp = request.getRemoteAddr();

if(this.ip.containsKey(requestIp)){

    noEsLaPrimeraVegada(out);
}
else{

    esLaPrimeraVegada(requestIp, out);
}
```

Amb l'objecte `request` obteniu l'adreça IP del client que ha fet la petició. Una vegada teniu l'adreça IP, comproveu si hi ha accedit amb anterioritat:

```
if(this.ip.containsKey(requestIp)){
```

**Objecte Java HashMap**

És una de les col·leccions de Java més populars entre els programadors. Existeixen altres col·leccions basades en Hash com són *HashTable i ConcurrentHashMap* però el seu rendiment és més baix en entorns amb un sol fil de processament. La col·lecció *HashMap* permet als programadors fer molts tipus d'operacions com afegir un element, treure'l, recórrer tots els elements, calcular la seva mida, obtenir totes les claus del *hash* o tots els seus valors, entre d'altres operacions interessants.

La variable `ip` és de tipus *HashMap*. Els objectes de tipus *HashMap* permeten emmagatzemar objectes del tipus `clau:valor`. L'avantatge d'aquests objectes és que recuperar un element, si se sap la clau, és molt ràpid. A més a més, us proporciona el mètode `containsKey(clau)`, que us retornarà *true* si la IP està guardada en el *HashMap* i *false* en cas contrari. Aquesta variable mai oblidarà aquest valor si no l'elimineu vosaltres, ja que l'objecte *servlet* perdurarà entre diferents peticions.

Si no es troba emmagatzemada vol dir que és la primera vegada que un usuari amb aquesta IP ha accedit a la pàgina. Llavors s'executarà la funció `esLaPrimeraVegada()`, que té el següent codi:

```
1 private void esLaPrimeraVegada(String requestIp, PrintWriter out){
2
3     this.ip.put(requestIp, "");
4
5     out.println("<h1>És la primera vegada que accedeixes a la pàgina. Benvingut
          .</h1>");
6     out.println("<p style='color:red;'>Accedeix al nostre patrocinador clicant
          al següent enllaç:</p>");
7     out.println("<a href='" + this.urlPublicitat + "'>Pàgina web del
          patrocinador</a>");
8 }
```

Primer de tot s'emmagatzema la IP dintre del *HashMap*: *this.ip.put(requestIP, "")*, i a continuació s'escriu el pedaç de pàgina corresponent als usuaris que hi accedeixen per primera vegada, com per exemple l'URL del patrocinador de la web.

Si, en canvi, no és la primera vegada que s'hi accedeix, llavors el codi anterior no s'executa. La funció que s'executaria seria `noEsLaPrimeraVegada();`, que té el següent codi:

```
1 private void noEsLaPrimeraVegada(PrintWriter out){
2     out.println("<h1>Gràcies per tornar a la pàgina web. Ja no veuràs el
          Patrocinador.</h1>");
3 }
```

La funció anterior només informa l'usuari que ja hi ha accedit prèviament i li dóna les gràcies per tornar a entrar-hi.

D'altra banda, si executeu el *servlet* Publicitat veureu que la variable `num` no canvia, ja que és un comptador que només s'actualitza dintre de la funció `init` i, en canvi, la variable `visites` s'actualitza cada vegada que hi ha una petició. Aquest és un símptoma que l'objecte *servlet* Publicitat és únic i s'executa el mètode `service` en cada petició.

### El mètode destroy del 'servlet' Publicitat

Durant la implementació d'aquest *servlet* no ha estat necessària la utilització de recursos que calgui eliminar utilitzant aquest mètode.

El mètode `destroy` es crida només una vegada al final del cicle de vida d'un *servlet*. Aquest mètode li dóna al seu *servlet* l'oportunitat de tancar les connexions

de base de dades, aturar subprocessos de fons o escriure llistes de *cookies*, així com fer altres activitats de neteja.

La definició del mètode `destroy` és la següent:

```
public void destroy ( ) {
    // La finalització de codi ...
}
```

### 1.3.2 Exemple de Publicitat amb redireccions

Es vol modificar el *servlet* Publicitat afegint-hi redireccions. En comptes d'escriure directament en la resposta de la petició es vol enviar l'usuari a una pàgina HTML.

En concret, si és la primera vegada que l'usuari accedeix al *servlet* es mostrarà la pàgina anomenada *1a_vegada.html*. La podeu crear accedint a *File / New File / HTML5 / HTML File*, i hi podeu afegir el següent codi:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Servlet Publicitat</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>És la primera vegada que accedeixes a la pàgina. Benvingut.</h1>
        <p style='color:red;'>Accedeix al nostre patrocinador clicant al següent enllaç:</p>
        <a href='http://ioc.xtec.cat'>Pàgina web del patrocinador</a><br>
        <a href='Publicitat2'>Tornar a accedir</a>
    </body>
</html>
```

En canvi, si no és la primera vegada l'usuari veurà el fitxer HTML anomenat resta_vegades.html. El podeu crear accedint a *File / New File / HTML5 / HTML File* i podeu afegir-hi el següent codi:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Servlet Publicitat</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>Gràcies per tornar a la pàgina web. Ja no veuràs el patrocinador.</h1>
        <a href='Publicitat2'>Tornar a accedir</a>

    </body>
</html>
```

Per fer aquest exercici creareu un *servlet* nou (*File / New File / Web / Servlet*) en el mateix projecte Maven de l'exemple anterior i l'anomenareu Publicitat2. Aquest cop, durant la creació del *servlet* no afegireu un paràmetre addicional, però sí hi la descripció del *servlet* al fitxer de configuració web.xml.

Fixeu-vos que en el codi afegit al fitxer de configuració web.xml ja no apareix cap paràmetre inicial:

```
...
<servlet>
     <servlet-name>Publicitat2</servlet-name>
     <servlet-class>cat.ioc.m7.servlets.Publicitat2</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>Publicitat2</servlet-name>
   <url-pattern>/Publicitat2</url-pattern>
</servlet-mapping>
...
```

Una possible implementació del *servlet* demanat és la següent:

```
public class Publicitat2 extends HttpServlet {

    private HashMap ip;

    @Override
    public void init (ServletConfig config) throws ServletException
    {
        super.init(config);
        this.ip = new HashMap();
    }

    @Override
    public void service(HttpServletRequest request, HttpServletResponse
         response) throws ServletException, IOException{

            String requestIp = request.getRemoteAddr();

            if(this.ip.containsKey(requestIp)){

                //redirect html - resta de vegades
                response.sendRedirect("resta_vegades.html");
            }
            else{

                //redirectHTML  -  1a vegada
                response.sendRedirect("1a_vegada.html");
                this.ip.put(requestIp, "");
            }
    }

}
```

Igual que abans, s'utilitza el mètode `init` per inicialitzar les variables privades. En aquest exemple s'han tret les variables de comptadors de visites i l'URL del patrocinador. En aquest cas, el patrocinador s'ha afegit directament en la pàgina redirigida.

El mètode `service` s'ha reduït considerablement. Ara ja no cal que s'escrigui la pàgina resultant, sinó que pot enviar l'usuari a una web o a una altra depenent de si és la primera vegada o no. La propietat `ip` permet fer aquesta distinció.

**sendRedirect versus forward**

S'utilitza el mètode *sendRedirect* quan es vol dirigir l'usuari a una altra pàgina d'un altre domini o servidor. En canvi, quan la redirecció es fa cap a una pàgina del mateix servidor o domini s'utilitza el mètode *forward*.

Per redirigir a una altra pàgina s'utilitza el mètode `sendRedirect` de l'objecte `response`. Aquesta funció redirigeix a un altre recurs, la qual cosa obliga el client (navegador) a fer automàticament una petició al nou recurs. Les redireccions poden ser internes (del mateix servidor) o externes (a un altre servidor). El client veurà a l'URL del navegador el nou recurs on s'ha accedit.

Un altre mètode per redireccionar és utilitzant el *request dispatching*. En aquest cas, en comptes d'utilitzar l'objecte `response` s'empra l'objecte `request` per reenviar la mateixa petició a un altre recurs. Així, el client no ha de demanar el nou recurs. De fet, no sabrà la pàgina li ha donat un altre recurs, ja que l'URL del navegador no canviarà.

Un exemple d'utilització del *request dispatching* és el següent:

```
RequestDispatcher rs = request.getRequestDispatcher("nouRecurs.html");
rs.forward(request,response);
```

Proveu de canviar l'exemple anterior utilitzant el mètode `getRequestDispatcher` de l'objecte `request`.

### 1.3.3 Exemple EndevinaColor

Intenteu canviar el *servlet* EndevinaColor creat anteriorment. Creeu un altre *servlet* anomenat EndevinaColor2 i afegiu-hi les següents modificacions:

- Si és la primera vegada, ha de mostrar el llistat d'enllaços amb els colors (el que abans era la pàgina EndevinaColor.html).
- Si és la segona vegada, ha de mostrar el resultat de la comparació, és a dir, informar si s'ha endevinat el color.

En aquest cas, en comptes de sobreescriure el mètode `service` pots utilitzar el mètode `processRequest` que crea l'IDE NetBeans.

```
private String colorInicial;

@Override
public void init (ServletConfig config) throws ServletException
{

    super.init(config);

    //Color configurat com a paràmetre inicial. És el color a endevinar.
    this.colorInicial = getServletConfig().getInitParameter("color");
}


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

    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {

        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Endevina el color</title>");
        out.println("</head>");
        out.println("<body>");

        String colorUsuari = request.getParameter("color");
```

```
28
29          if(colorUsuari != null && !colorUsuari.equals("") ){
30
31              crearPaginaGuanyador(colorUsuari, out);
32
33          }
34          else{
35
36              crearPaginaInicial(out);
37          }
38
39          out.println("</body>");
40          out.println("</html>");
41
42      }
43  }
44  ...
```

Fixeu-vos que s'ha sobreescrit el mètode `init` per obtenir la configuració inicial del *servlet*. En concret, s'ha recuperat el color configurat en el fitxer web.xml. Recordeu que és aquest el color que s'ha d'endevinar.

El mètode `processRequest` és el mètode que porta a terme la mateixa funció que `service`. Així, aquí farem tot el codi que s'ha d'executar a cada petició d'un client.

Bàsicament, es crea una resposta depenent de si és la primera vegada que s'hi accedeix o no. Per determinar si és o no la primera vegada s'accedeix al *request* de la petició cercant el color que proposa l'usuari. Si no proposa cap color s'executa la funció `crearPaginaInicial()`; si, en canvi, l'usuari proposa un color, es mira si ha guanyat o no amb la crida de la funció `crearPaginaGuanyador()`.

El codi de les funcions és el següent:

```
1   private void crearPaginaGuanyador(String colorUsuari, PrintWriter out){
2
3       String endevinat = "Llàstima, has perdut!";
4
5       if(this.colorInicial.toLowerCase().equals(colorUsuari.toLowerCase())){
6           endevinat = "Felicitats! Has endevinat el color.";
7       }
8
9       out.println("<h1>" + endevinat +"</h1>");
10      out.println("<a href='EndevinaColor2'>Tornar<a/>");
11
12  }
13
14  private void crearPaginaInicial(PrintWriter out) {
15      out.println("<h1>Endevina el color configurat:</h1>");
16      out.println("<a href='EndevinaColor2?color=white'>blanc</a>");
17      out.println("<a href='EndevinaColor2?color=red'>vermell</a>");
18      out.println("<a href='EndevinaColor2?color=blue'>blau</a>");
19      out.println("<a href='EndevinaColor2?color=yellow'>groc</a>");
20      out.println("<a href='EndevinaColor2?color=green'>verd</a>");
21      out.println("<a href='EndevinaColor2?color=black'>negre</a>");
22  }
```

Ara ja no us cal cap pàgina HTML addicional. El mateix *servlet* crea la pàgina que necessita veure l'usuari depenent de la petició que ha fet.

## 1.4 Què s'ha après

En aquest apartat l'alumne ha après:

- La creació i configuració d'un *servlet*.
- El cicle de vida d'un *servlet* (`ini`, `service`, `destroy`).
- Classes associades als *servlets*, com poden ser `ServletContext`, `HttpServletRequest` i `HttpServletResponse`.
- La redirecció d'una petició utilitzant la mateixa petició original o creant-ne una de nova.

Per aprofundir en aquests llenguatges es recomana la realització de les activitats associades a aquest apartat. Una vegada realitzades, l'alumne estarà preparat per continuar amb l'aprenentatge dels *servlets*.

## 2. Formularis amb 'servlets' i EJB

En aquest apartat s'explicarà com enviar informació des d'un formulari web a un *servlet*. Aprendrem a obtenir aquesta informació, tractar-la i enviar una pàgina de resposta a l'usuari.

Ens endinsarem en l'aprenentatge de l'arquitectura **Enterprise JavaBeans (EJB)**. Veurem com funciona, quins elements la componen i com interacciona amb l'arquitectura Java Web. Així, aprendrem a enviar i rebre informació entre els formularis, els *servlets* i els components EJB.

També aprendrem una tècnica per comprovar el format de les dades que ens envia l'usuari amb un formulari. Aquesta tècnica utilitza patrons i expressions regulars que es poden afegir, per automatitzar la comprovació de les dades, als components EJB.

Tots aquest conceptes s'explicaran partint de l'exemple. Començarem amb un exemple senzill per calcular el sou d'una família, i les dades s'enviaran a un *servlet*. Continuarem amb l'elaboració d'un blog on introduïren l'arquitectura EJB. Acabarem l'apartat demanant les dades d'un usuari per inscriure'l en l'aplicació, on veurem com tractar les dades i automatitzar el procés per comprovar les restriccions requerides pel programa.

Podeu descarregar-vos el codi Java, que utilitzarem en aquest apartat, des de l'enllaç que trobareu a l'apartat d'annexos de la unitat. Recordeu que podeu utilitzar la funció d'importar del Netbeans per accedir a aquest contingut.

Per poder fer aquest apartat es recomana haver fet l'apartat "*Servlets*", on se n'explica el funcionament i el seu cicle de vida.

### 2.1 Calculant el sou net

En aquest apartat aprendreu a utilitzar els *servlets* com a receptors de les peticions `GET` o `POST` que s'envien des dels formularis HTML. Utilitzareu els formularis per enviar informació des del navegador fins al *servlet*.

**GET versus POST**

El mètode **GET** és un mètode del protocol HTTP que envia la informació al servidor utilitzant la URL. En canvi, el mètode **POST** envia la informació utilitzant la capçalera de la petició i, per tant, sense embrutar la URL en el navegador.

Comenceu creant un nou projecte amb Maven (*File / New Project / Maven / Web Application*) i l'anomeneu formServlets. Posteriorment creareu un *servlet* anomenat CalculSouServlet. Recordeu que per crear un *servlet* amb NetBeans heu d'accedir a *File / New File / Web / Servlet*. No cal que afegiu la configuració del *servlet* al fitxer web.xml. En aquest cas, utilitzareu les anotacions per configurar-lo.

Aquest *servlet* calcularà el sou net que rebrà una família a final de mes tenint en compte el nombre de fills i el sou brut que es cobra. Suposeu que la retenció normal és del 21%, i que per cada fill que es tingui es rebaixarà 5 punts aquest percentatge.

Segons l'enunciat, necessiteu que cada família que vulgui calcular el sou net enviï al *servlet* les següents dades:

- sou brut
- nombre de fills

Necessiteu un formulari HTML amb els elements necessaris per demanar aquesta informació. Creareu una nova pàgina web (*File / New File / Html5 / Html File* ) amb el nom calcularSou.html. El contingut de la pàgina és el següent:

```
<!DOCTYPE html>
<html>
    <head>
        <title>form-servlet</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>Càlcul del salari:</h1>
        <form method="GET" action="CalculSouServlet">
            <label for="salari">Salari brut:</label>
            <input id="salari" type="number" name="salariBrut" min="0"/>
            <label for="fill">Nombre de fills:</label>
            <input id="fill" type="number" name="fills" min="0">
            <input type="submit">
        </form>
    </body>
</html>
```

Per elaborar la pàgina calcularSou.html s'ha creat un formulari web on s'han creat dos camps (*inputs*) que permeten a l'usuari introduir informació. El primer camp correspon al salari brut que cobra la família:

```
<input id="salari" type="number" name="salariBrut" min="0"/>
```

Aquest camp del formulari és de tipus numèric, i com a mínim el valor que introdueixi l'usuari ha de ser 0, així es fa un petit control d'errors abans d'enviar qualsevol dada al *servlet*. Quan l'usuari introdueixi el seu sou, aquest s'enviarà amb l'identificador que surt al costat de l'atribut `name`.

El segon camp correspon al nombre de fills que té la família:

```
<input id="fill" type="number" name="fills" min="0">
```

Igual que s'ha fet abans, també es fa un petit control d'errors. El tipus de dada que s'ha de posar és numèrica i més gran o igual que 0. Quan l'usuari introdueixi un valor, aquest s'enviarà al *servlet* amb el nom *fills*.

Una vegada s'han definit els camps que s'han d'enviar al servidor cal configurar el mètode d'enviament de la informació, així com el recurs que la rebrà. Aquesta informació es configura en la mateixa etiqueta `form`. El mètode pot ser `POST` o `GET`, i el recurs que rebrà la informació ha de ser el *servlet* CalculSouServlet.

```
<form method="GET" action="CalculSouServlet">
```

Reviseu que teniu una anotació al *servlet* (fitxer CalculaSouServlet.java) que indica que escolta les peticions enviades a l'URL CalculSouServlet. L'anotació ha de ser semblant a aquesta:

```
@WebServlet(name = "CalculSouServlet", urlPatterns = {"/CalculSouServlet"})
```

El nom del recurs configurat a la propietat `urlPatterns` del *servlet* ha de coincidir amb el nom de la propietat `action` configurada al formulari.

Una vegada arribats a aquest punt, ja teniu la part del client acabada, i continuareu amb la part del servidor. Ara calcularеu el sou net que percebrà una família segons les dades que hagi enviat amb el formulari.

Modificareu la funció `processRequest` del *servlet*. Aquesta funció s'executa sempre que es rebi una petició `POST` o `GET` indistintament. El contingut de la funció és el següent:

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
    response)
        throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet formServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Dades rebudes del formulari</h1>");

        int brut = Integer.parseInt(request.getParameter("salariBrut"));
        out.println("<p>Salari brut: " + brut + "</p>");

        int fills = Integer.parseInt(request.getParameter("fills"));
        out.println("<p>Nombre de fills: " + fills + "</p>");

        out.println("<h1>Càlcul del salari net:</h1>");
        int retencio = 21 - (5*fills);

        // Fórmula per calcular la retenció: k = (int)(value*(percentage/100.0f
            ));
        int net = brut -((int)(brut * (retencio/100.0f)));

        out.println("<p>Tens una retenció del " + retencio + " per cent</p>");
        out.println("<p>El teu salari net és de " + net + " euros</p>");
        out.println("<a href='calcularSou.html'> Tornar </a>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Bàsicament, la funció `processRequest` obté els paràmetres que ha enviat l'usuari i fa el càlcul de la retenció per poder mostrar el salari net. Vegeu com s'obté el paràmetre `salariBrut`:

```
int brut = Integer.parseInt(request.getParameter("salariBrut"));
```

Tota la informació que s'envia des del navegador cap al *servlet* s'emmagatzema a l'objecte `request` (petició). Aquest objecte té un mètode anomenat `.getParameter(name)` per obtenir les dades que ha enviat l'usuari. El nom que

s'ha d'utilitzar per obtenir les dades és el nom que s'ha configurat a la propietat `name` de l'*input* corresponent. Per exemple, si voleu obtenir el salari brut hem de veure com s'ha creat el formulari:

```
<input id="salari" type="number" name="salariBrut" min="0"/>
```

El nom configurat és `salariBrut`. Aquest nom és el que s'ha d'utilitzar al mètode `.getParameter` de l'objecte `request` per obtenir la dada introduïda:

```
request.getParameter("salariBrut");
```

Finalment, com que voleu poder operar amb aquest número s'ha de convertir en un *integer*, ja que el tipus de dada que retorna el mètode `getParametre` és de tipus *string*.

Feu la mateixa operació amb el paràmetre que representa el nombre de fills.

```
int fills = Integer.parseInt(request.getParameter("fills"));
```

Observeu que ambdós paràmetres s'han emmagatzemat en dues variables, anomenades *brut* i *fills*, de tipus *integer*, per poder operar amb elles i calcular el sou net.

El sou net es calcula multiplicant el salari brut per la retenció apropiada: *(net = brut*(percentage/100))*. Primer s'ha de calcular quina retenció s'ha d'aplicar.

```
int retencio = 21 - (5*fills);
```

La retenció màxima s'estableix, segons l'enunciat, en el 21%. Per cada fill que tingui la família s'han de treure 5 punts de retenció. Per tant, amb aquesta fórmula: *retenció = retencioMaxima - (numFills*5)* obteniu la retenció real que s'ha d'aplicar.

Una vegada s'ha calculat la retenció real es pot aplicar a la fórmula per calcular el sou net:

```
int net = brut -((int)(brut * (retencio/100.0f)));
```

Tingueu en compte que a Java, per fer el càlcul dels percentatges, les dades han d'estar en *float*. Si convertiu el valor 100 a *float*, la seva divisió amb un *integer* també dóna un *float* i obtindreu correctament aquest percentatge.

Finalment, es mostren les dades per pantalla amb el càlcul del sou net i es permet tornar a la pàgina principal per si es vol fer un altre càlcul.

## 2.2 Escrivint un 'post'

En aquest apartat s'explicarà la validació de dades d'un formulari utilitzant *servlets* i EJB. Aprendrem a integrar aquests elements perquè funcionin plegats

validant dades d'un formulari. Es veuran diverses estratègies de funcionament de l'arquitectura EJB per fer-ne la validació.

Es vol crear una aplicació que permeti escriure un missatge per pantalla que no ha de superar els 150 caràcters. Per permetre escriure un missatge, l'aplicació demana un correu electrònic vàlid i la data de naixement de la persona. Si la persona té menys de 18 anys no se li permet escriure el missatge.

Per fer aquest apartat aprofitareu el mateix projecte que a l'apartat anterior. Creeu una nova pàgina HTML, amb el programa NetBeans, anomenada escriurePost.html (*File / New File / HTML5 / HTML File*), on afegireu les dades a un formulari web HTML. El codi de la pàgina pot ser semblant a aquest:

```
<!DOCTYPE html>
<html>
    <head>
        <title>EJB-SERVLET</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
      <h1>Escriu un missatge:</h1>
        <form id="formulari" method="POST" action="PostServlet">
            <label for="mail">Correu electrònic:</label>
            <input id="mail" type="text" name="mail"/>
            <label for="age">Edat:</label>
            <input id="age" type="number" name="edat" min="0">
            <input type="submit">
        </form>
        <label>Missatge:</label>
        <textarea name="missatgePost" form="formulari"></textarea>
    </body>
</html>
```

La pàgina HTML consta d'un formulari on es demana a l'usuari tres dades:

- adreça de correu electrònic (*name=mail*)
- edat de la persona (*name=edat*)
- missatge o *post* a escriure per pantalla (*name=missatgePost*)

Per poder escriure el missatge per pantalla, les dades s'han de validar. Les validacions són les següents:

- La longitud del missatge (*missatgePost*) ha d'estar entre 0 i 150 caràcters.
- L'edat de la persona ha de ser major o igual a 18 anys.
- El correu electrònic ha de ser vàlid.

Per implementar la validació de les dades utilitzareu l'arquitectura EJB, que permet la reutilització dels seus components en aquesta o en altres aplicacions.

Un **Enterprise JavaBean** (EJB) és una arquitectura de components de servidor que simplifica el procés de construcció d'aplicacions de components empresarials distribuïts en Java.

S'utilitza l'arquitectura EJB sempre que vulgueu:

- Que l'aplicació sigui escalable. Potser a mesura que el nombre d'usuaris vagi creixent es necessitarà distribuir l'aplicació en diferents màquines.
- Integritat de dades mitjançant transaccions. Els components EJB poden crear transaccions i tenen mecanismes per accedir concurrentment a objectes compartits.
- Que l'aplicació tingui una gran varietat de clients. Amb poques línies de codi, clients remots poden localitzar fàcilment Enterprise Beans.

Els components EJB no es poden utilitzar directament per interaccionar amb el protocol HTTP. Necessiten un *framework*, com pot ser Spring, JSF o aplicacions basades en *servlets*, per tractar amb el protocol HTTP i després enviar la informació necessària als components EJB.

Els *frameworks* faciliten al programador la seva tasca donat-li una metodologia estàndard, una organització del projecte i uns recursos propis que moltes vegades fan molt fàcil la tasca de programar grans aplicacions. Alguns *frameworks* que existeixen per a Java son els següents: JSP, JSF, Spring, Struts, Tapestry, Googgle Web Toolkit (GWT), entre d'altres.

Així, necessiteu crear un *servlet* que rebi les dades enviades amb el formulari i que després les enviï a un component EJB per a la seva validació.

Creeu un *servlet* nou anomenat PostServlet (*File / New File / Web / Servlet* ). No cal que afegiu la configuració del *servlet* al fitxer web.xml. En aquest cas, utilitzareu les anotacions per configurar-lo.

El *servlet* ha de portar a terme tres tasques:

- Obtenir les dades que ha enviat l'usuari
- Enviar les dades a un component EJB per validar-les
- Mostrar per pantalla el resultat de la validació

El codi corresponent a aquest funcionament és el següent:

```
1  @WebServlet(name = "PostServlet", urlPatterns = {"/PostServlet"})
2  public class PostServlet extends HttpServlet {
3
4      @EJB
5      private PostBeanLocal validation;
6
7      protected void processRequest(HttpServletRequest request,
           HttpServletResponse response)
8              throws ServletException, IOException {
9          response.setContentType("text/html;charset=UTF-8");
10         try (PrintWriter out = response.getWriter()) {
11             out.println("<!DOCTYPE html>");
12             out.println("<html>");
13             out.println("<head>");
14             out.println("<title>Servlet ServletValidation</title>");
15             out.println("</head>");
16             out.println("<body>");
17             out.println("<h1>EJB validation</h1>");
18
19             String mail = request.getParameter("mail");
20             if (validation.isValidEmail(mail)) {
21                 out.println("<p>El correu electrònic " + mail + " és vàlid</p>"
                       );
22             } else {
```

```
23                 out.println("<p>El correu electrònic no és vàlid.</p>");
24             }
25
26             String edat = request.getParameter("edat");
27             if (validation.isValidAge(edat)) {
28                 out.println("<p>Tens" + edat + " anys i pots escriure un
                       missatge a l'aplicació.</p>");
29             } else {
30                 out.println("<p>Has de ser major d'edat per escriure un
                       missatge a l'aplicació.</p>");
31             }
32
33             String missatge = request.getParameter("missatgePost");
34             if (validation.isValidPost(missatge)) {
35                 out.println("<p>El missatge '" + missatge + "' és vàlid.</p>");
36             } else {
37                 out.println("<p>El missatge no és vàlid. Ha de tenir menys de
                       150 caràcters.</p>");
38             }
39
40             out.println("</body>");
41             out.println("</html>");
42         }
43     }
```

Com podeu observar, en el *servlet* PostServlet es repeteix el funcionament esperat per a cadascuna de les dades enviades des del formulari.

Primer es recupera una de les dades enviades, per exemple l'edat:

```
1 String edat = request.getParameter("edat");
```

A continuació, es valida l'edat:

```
1 if (validation.isValidAge(edat)) {
```

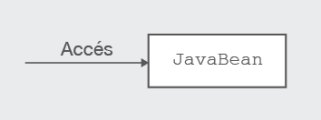
Segons el resultat de la validació, s'envia un missatge o un altre a l'usuari:

```
1 if (validation.isValidAge(edat)) {
2     out.println("<p>Tens" + edat + " anys i pots escriure un missatge a l'
          aplicació.</p>");
3 } else {
4     out.println("<p>Has de ser major d'edat per escriure un missatge a l'
          aplicació.</p>");
5 }
```

El codi que s'ha d'introduir en el *servlet* s'ha reduït molt. Ara el *servlet* només conté la part visual de la pàgina de resposta. Tota la lògica de l'aplicació ha estat externalitzada a un component EJB.

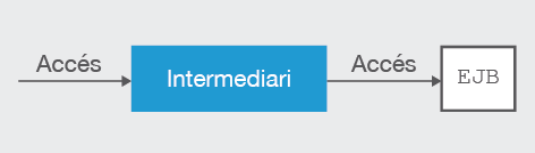
Un EJB (Enterprise JavaBean) és un component que ha d'executar-se en un contenidor d'EJB, i es diferencia molt d'un JavaBean normal. Un JavaBean és un objecte al qual accediu de manera directa des de la vostra aplicació (vegeu la figura 3.1).

**FIGURA 2.1.** Accés a un JavaBean



En canvi, un EJB és un component al qual no podeu accedir d'una forma tan directa i sempre hi accediu a través d'algun tipus d'intermediari (vegeu la figura 3.2).

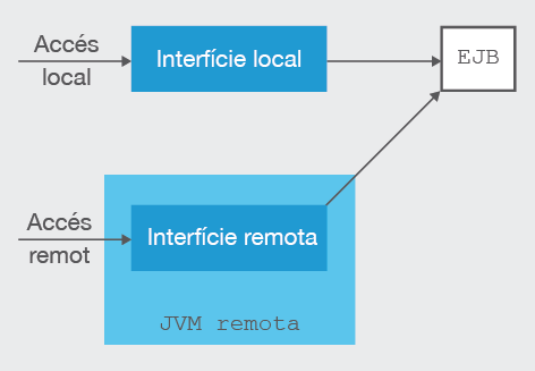
**Figura 2.2.** Accés a un EJB




Aquest intermediari aportarà una sèrie de serveis definits pels estàndards en els quals l'EJB es pot recolzar. Existeixen dos tipus d'intermediaris, l'intermediari local i l'intermediari remot.

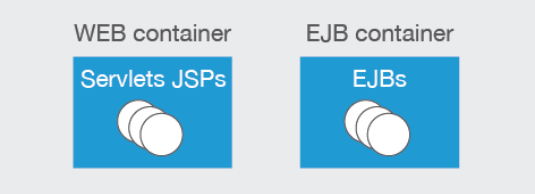
L'intermediari local és aquell que s'executa en la mateixa màquina virtual de Java que l'EJB. En canvi, el remot és aquell intermediari que s'executa en una màquina virtual de Java diferent de la màquina on s'executa l'EJB. Gràcies a aquests intermediaris es poden construir aplicacions distribuïdes (vegeu la figura 3.3).

**Figura 2.3.** Arquitectura EJB distribuïda




Cadascuna d'aquestes interfícies intermediàries s'encarrega de definir els mètodes que estaran a la disposició dels clients que els invoquen. Aquestes interfícies són les encarregades de donar accés als EJB a tots els serveis addicionals que suporta el contenidor EJB (vegeu la figura 3.4), com són el maneig de transaccions, la concurrència, la seguretat, la gestió de recursos, els serveis de xarxa, etc.

**Figura 2.4.** Contenidor d'EJB




Un **contenidor EJB (EJB Container)** és l'encarregat d'administrar l'execució dels components EJB. Controla totes les funcionalitats d'un EJB dintre del servidor actuant com a intermediari entre el servidor i l'aplicació.

Un contenidor d'EJB pot administrar dos tipus de components EJB: els *beans* de sessió (*Session Beans*) o els *beans* dirigits per missatges (*message-driven beans*).

Un contenidor web també és conegut com a Contenidor Servlet perquè la seva funció principal és administrar el cicle de vida dels *servlets*. Amb les aplicacions EJB van sorgir els seus contenidors propis per administrar els EJB.

Els *beans* de tipus ***message-driven*** processen missatges asíncrons. Els missatges es reben i s'envien mitjançant el servei de missatges de Java (JMS: Java Message Service).

Els ***beans* de sessió** encapsulen la lògica de l'aplicació i poden ser invocats per un client local, remot o un client d'un servei web (*web service*). El client ha d'accedir directament al mètode que vol executar del *bean*. El contenidor EJB s'encarrega de crear el *bean* corresponent i facilitar la interacció amb l'aplicació. Els *beans* de sessió no són persistents, és a dir, no s'emmagatzemen a la base de dades.

Si observeu el codi on executem la validació de l'edat no creem el *bean validation* (igual que no creem el *servlet*). El contenidor EJB crea per a nosaltres un *bean* que executarà la funcionalitat demanada.

```
1 if (validation.isValidAge(edat)) {
2     out.println("<p>Tens" + edat + " anys i pots escriure un missatge a l'
          aplicació.</p>");
3 } else {
4     out.println("<p>Has de ser major d'edat per escriure un missatge a l'
          aplicació.</p>");
5 }
```

Existeixen tres tipus de *beans* de sessió:

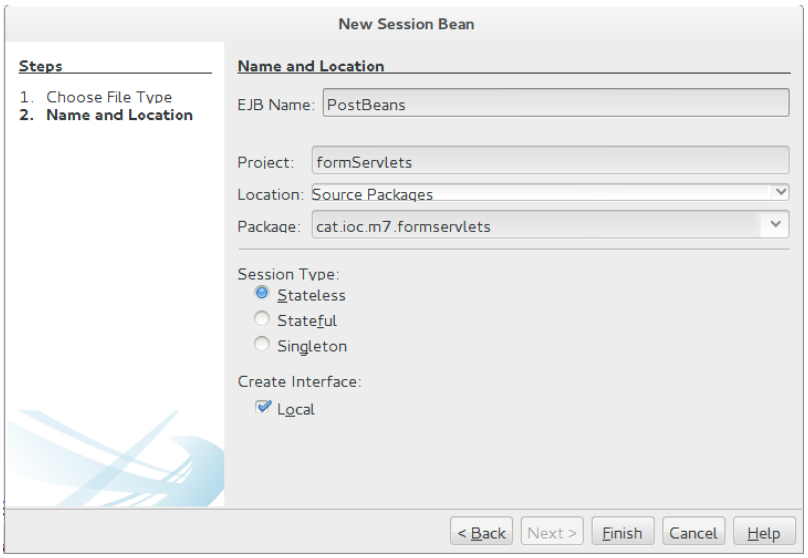
- *Beans* sense estat (***stateless***): no es modifiquen amb les crides dels clients. Els mètodes es posen a disposició de les aplicacions clients que els executen enviant dades i rebent resultats, però que no modifiquen internament el *bean*. Això permet al contenidor tenir un conjunt d'instàncies (*pool*) del mateix *bean* i anar assignant qualsevol instància del *bean* a qualsevol client. Fins i tot pot tenir la mateixa instància assignada a diferents clients, ja que l'assignació només dura el temps d'execució del mètode.

- *Beans* amb estat (***stateful***): aquests *beans* poden tenir propietats. Cada *bean* emmagatzema les dades d'un client durant la seva interacció i no es pot compartir entre diferents clients. Aquestes propietats es poden modificar quan el client va fent les crides als mètodes del *bean*. Aquestes dades no es guarden quan el client finalitza la sessió. En acabar la comunicació, el *bean* s'esborra.

- *Bean* ***singleton***: és un *bean* que s'instancia una vegada i existeix durant tota vida de l'aplicació.

En aquesta ocasió s'utilitza, a la vostra aplicació, un *bean* EJB sense estat. Per crear-lo aneu a *New File / Enterprise JavaBeans / Session Bean* (vegeu la figura 3.5). Anomeneu-lo PostBean i activeu la creació de la interfície local.

**FIGURA 2.5.** Nou 'bean' de sessió




Existeixen moltes maneres de fer la validació de les dades que introdueix l'usuari. Per exemple, HTML5 i Javascript, que són llenguatges del costat client, també proporcionen patrons per poder validar-les.

Per fer la validació del correu electrònic, de l'edat i del missatge enviat es necessiten tres funcions, una per cada valor que es vol validar. Llavors creem aquestes funcions a la interfície local creada. El seu nom és el nom del *bean* de sessió creat acabat en "Local" (PostBeanLocal). Si heu activat l'opció de creació de la interfície quan heu creat el *bean* de sessió, aquesta ja estarà creada. El codi de la interfície és el següent:

```
1  package cat.ioc.m7.formservlets;
2
3  import javax.ejb.Local;
4
5  @Local
6  public interface PostBeanLocal {
7
8      public Boolean isValidEmail(String email);
9
10     public Boolean isValidAge(String age);
11
12     public Boolean isValidPost(String message);
13
14 }
```

Fixeu-vos que l'anotació `@Local` s'utilitza per informar el contenidor d'EJB que correspon a una interfície local d'un EJB, és a dir, que una aplicació ubicada a la mateixa màquina virtual de Java, per exemple un *servlet*, que vulgui obtenir un PostBean ha d'utilitzar aquesta interfície per arribar als seus mètodes.

Aquestes mètodes es podran cridar de manera independent i el *bean* que les executi no emmagatzemarà cap informació. Així, pot haver-hi un *pool* de *beans* i en cada petició s'agafarà un *bean* qualsevol.

La classe `PostBean` és el *bean* de sessió que implementarà la interfície PostBeanLocal i, en conseqüència, les tres funcions anteriors. Modificareu aquest objecte amb la implementació de les tres funcions. Vegeu-ne una possible solució:

```
1  @Stateless
2  public class PostBean implements PostBeanLocal {
3
4      @Override
```

```
5       public Boolean isValidEmail(String email) {
6           //pattern
7           String regex = "^(.+)@(.+)$";
8           Pattern pattern = Pattern.compile(regex);
9           Matcher matcher = pattern.matcher(email);
10          return matcher.matches();
11      }
12
13      @Override
14      public Boolean isValidAge(String age) {
15          //min 18
16          if(age != null && !age.equals("")) return Integer.parseInt(age) >= 18;
17          return false;
18      }
19
20      @Override
21      public Boolean isValidPost(String message) {
22          //length post < 150
23          return message.length() <= 150;
24      }
25  }
```

El *bean* PostBean utilitza l'anotació `@Stateless`, que indica que és un *bean* sense estat. El contenidor EJB, automàticament, crea les interfícies associades llegint aquest atribut durant el seu desplegament en el servidor.

Observeu que PostBean implementa la interfície PostBeanLocal i, per tant, ha de sobreescriure les seves funcions. Amb l'anotació `@Override` s'informa el compilador de Java que el mètode està sobreescrit i aquest comprova que el mètode de la interfície correspongui al mètode del *bean*.

La implementació de les funcions de validació de l'edat i del missatge no tenen cap misteri. En el primer cas, es retorna *true* si l'edat és més gran o igual a 18, i *false* en cas contrari. En el segons cas, es calcula la longitud del missatge amb el mètode `.length()` i es compara amb 150. Retorna *true*, indicant que és més petit i que té la longitud correcta, i *false* en cas contrari.

La implementació de la funció de validació del correu electrònic s'ha realitzat amb un patró (*pattern*). Si el correu electrònic coincideix amb el patró retorna *true*, i retorna *false* en cas contrari.

Un **patró** (*pattern*) és un objecte que conté la representació d'una expressió regular.

Un **expressió regular** és una seqüència de caràcters que ajuden a trobar cadenes dintre de cadenes utilitzant una sintaxi especial. Es poden utilitzar per cercar o manipular cadenes de text.

La expressió regular utilitzada és *^(.+)@(.+)$*. Intentem desxifrar-la:

- Un punt (.) simbolitza qualsevol caràcter.
- El símbol "+" indica que es pot repetir l'expressió anterior una o més vegades.

- El símbol "@" és ell mateix. Indica que ha d'aparèixer.
- El símbol "^" indica que l'expressió següent ha d'aparèixer al principi de l'*string*.
- El símbol "$" indica que l'expressió anterior ha d'aparèixer al final de l'*string*.

En altres paraules, l'expressió regular "^(.+)@(.+)$" significa: al principi de l'*string* ha d'aparèixer com a mínim un caràcter, després hi ha d'haver una "@", i finalment ha d'acabar amb un caràcter o més.

No és un patró molt acurat per detectar correus electrònics vàlids i, per això, cal que intenteu elaborar un patró una mica més exacte amb aquestes opcions:

- L'expressió "\d" vol dir que ha d'aparèixer un dígit. És equivalent a [0-9].
- L'expressió "\D" vol dir que no ha de ser un dígit. És equivalent a [^0-9].
- L'expressió "\s" vol dir que hi ha d'haver un espai en blanc. És equivalent a [\t\n\x0b\r\f].
- L'expressió "\S" vol dir que no hi ha d'haver un espai en blanc. És equivalent a "[^\s]".
- L'expressió "\w" vol dir que hi ha d'haver una lletra majúscula, minúscula, un dígit o el caràcter "_". És equivalent a "[a-zA-Z0-9_]".
- L'expressió "\W" vol dir que no hi ha d'haver una lletra majúscula, minúscula, un dígit o el caràcter "_". És equivalent a "[^/w]".
- L'expressió "\b" estableix el límit d'una paraula.
- El símbol "*" indica que una expressió es pot repetir 0 o més vegades.
- El símbol "?" indica 0 o 1 vegades.

Exemples:

- El símbol "." vol dir qualsevol caràcter.
- L'expressió "[^abc]" vol dir que ha de contenir qualsevol símbol excepte "a", "b" o "c". El símbol "^" dintre dels claudàtors indica negació.
- L'expressió "[a-z1-9]" indica un rang. Les lletres minúscules des de la 'a' fins a la "z" (incloent-hi ambdues) i els dígits 1 fins al 9 (també incloent-hi ambdues).

A Java no podeu utilitzar una sola "\", s'han d'utilitzar dues. Així, si volem utilitzar l'expressió "\d" hauríem de posar "\\d".

Una vegada s'ha elaborat l'expressió regular, aquesta s'ha de compilar en un *pattern*. L'únic que falta és poder comparar-la amb una cadena de text per veure si aquesta compleix l'expressió regular. La comparació, a Java, la fa un objecte anomenat comparador (`matcher`).

Un objecte de tipus `matcher` és una eina que interpreta un *pattern* i fa una **operació de comparació** sobre una cadena de text donada.

Molts processadors de texts i llenguatges de programació fan ús d'expressions regulars per a procediments de cerca o bé de cerca i substitució de texts.

Vegeu com treballen conjuntament l'expressió regular, l'objecte `pattern` i l'objecte `matcher` per validar una adreça de correu electrònic:

```
String email = "a@a";
String regex = "^(.+)@(.+)$";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(email);
return matcher.matches();
```

Vegeu ara la utilització del *bean* de sessió sense estat al *servlet*. Vegeu com es declara el *bean*:

```
@WebServlet(name = "PostServlet", urlPatterns = {"/PostServlet"})
public class PostServlet extends HttpServlet {

    @EJB
    private PostBeanLocal validation;

    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet ServletValidation</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>EJB validation</h1>");

            String mail = request.getParameter("mail");
            if (validation.isValidEmail(mail)) {
                out.println("<p>El correu electrònic " + mail + " és vàlid</p>"
                    );
            } else {
                out.println("<p>El correu electrònic no és vàlid.</p>");
            }

            String edat = request.getParameter("edat");
            if (validation.isValidAge(edat)) {
                out.println("<p>Tens" + edat + " anys i pots escriure un
                    missatge a l'aplicació.</p>");
            } else {
                out.println("<p>Has de ser major d'edat per escriure un
                    missatge a l'aplicació.</p>");
            }

            String missatge = request.getParameter("missatgePost");
            if (validation.isValidPost(missatge)) {
                out.println("<p>El missatge '" + missatge + "' és vàlid.</p>");
            } else {
                out.println("<p>El missatge no és vàlid. Ha de tenir menys de
                    150 caràcters.</p>");
            }

            out.println("</body>");
            out.println("</html>");
        }
    }
```

El *bean* forma part del *servlet* i es defineix com una propietat de la classe. Com veieu, no podeu utilitzar directament el *bean*, s'ha d'emprar la seva interfície local, ja que el *servlet* s'executarà a la mateixa màquina virtual que el *bean* de sessió.

La seva declaració com a propietat del *servlet* és la següent:

```
1 @EJB
2 private PostBeanLocal validation;
```

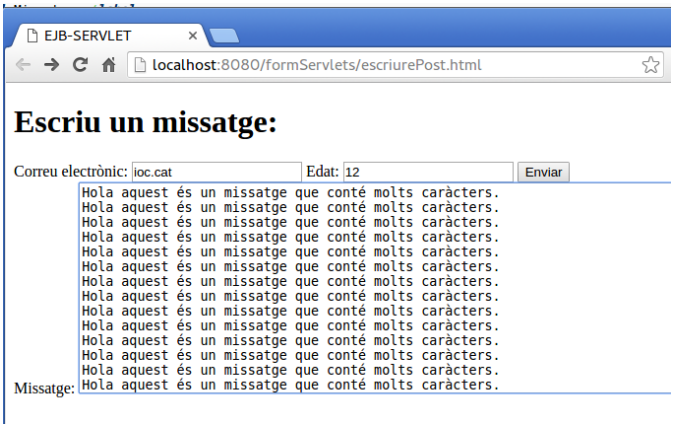
Fixeu-vos en l'anotació `@EJB`. Vol dir que és un *bean* del tipus EJB i que la seva creació li pertany al contenidor de *beans* EJB.

Quan s'utilitzi la propietat `validation`, el contenidor EJB agafarà un *bean* del *pool* de *beans* que té disponibles per fer l'execució del mètode utilitzat. Per a cada execució de cadascun dels mètodes fa la mateixa operació. No es pot assegurar que sempre sigui el mateix *bean*.

Aquesta operació s'anomena **injecció**. El contenidor d'Enterprise JavaBeans subministra un *bean* quan una aplicació li demana un *bean* del seu *pool* de *beans*.

Vegeu el funcionament de l'aplicació Escriure un Post. A la figura 3.6 podeu veure com ens mostra un formulari per introduir les dades següents: edat, correu electrònic i el missatge.

**FIGURA 2.6.** Pantalla inicial de l'aplicació Escriure un Post




A la següent pantalla, les dades es validen amb EJB i es mostra el resultat de la validació, com podeu veure a la figura 3.7:

**FIGURA 2.7.** Pantalla de validació de l'aplicació Escriure un Post

### 2.2.1 'Beans' de sessió amb estat

Vegeu ara una altra implementació de l'exercici Escriure un Post. Aquesta vegada s'utilitzarà un *bean* amb estat (*stateful*).

Els *beans* amb estat poden tenir propietats i, per tant, el contenidor EJB no tindrà un *pool* d'aquests tipus de *beans*. El contenidor d'EJB ens crearà el *bean* però ens assegura que mentre duri la petició aquest *bean* és nostre i podem utilitzar-lo per guardar informació a les seves propietats.

L'aplicació necessita tres dades de l'usuari: el correu electrònic, l'edat de l'usuari i el missatge a publicar. Aquestes dades corresponen a la informació que guardarà el *bean*.

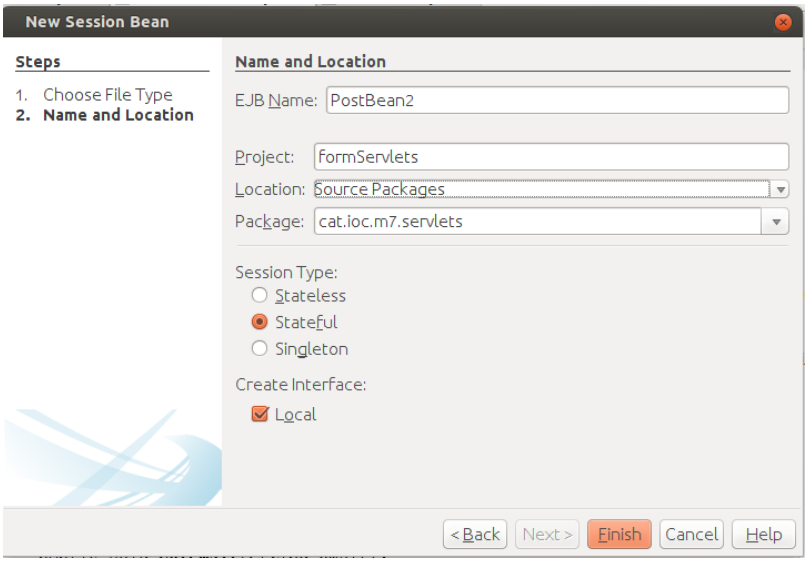
L'objectiu de l'exercici és validar les dades. En comptes d'utilitzar tres mètodes per validar-les emprarem les validacions de Java del *package validation*.

**JavaBeans Validation** (*Bean Validation*) és un model de validació basat en restriccions (*constraints*). Les restriccions es posen com a anotacions en una propietat, un mètode o una classe `JavaBean`.

Les restriccions (***constraints***) les pot definir l'usuari (*custom constraint*), o bé utilitzar les que porta per defecte el model (*built-in constraints*).

Començareu creant el *bean* EJB amb estat ( *File / New File / Enterprise JavaBeans/ Session Bean*) i utilitzareu el mateix projecte Maven que heu emprat fins ara. La figura 3.8 mostra la creació del *bean* amb estat:

**Figura 2.8.** Creació d'un EJB 'stateful bean'




El *bean* amb estat necessita, igual que el *bean* sense estat, d'una interfície local per accedir al *bean*. Activareu la creació automàtica d'aquesta interfície quan creeu el *bean*.

Aquesta interfície ha de tenir els mètodes que s'utilitzaran del *bean*, no les seves propietats. Així, la interfície, en aquest cas, tindrà els *setters* i *getters* de les propietats. Aquests mètodes se sobreescriuran al *bean* amb estat.

De moment, el codi de la interfície local és el següent:

```
package cat.ioc.m7.formservlets;

import javax.ejb.Local;

@Local
public interface PostBean2Local {

    public int getEdat();

    public String getMessage();

    public String getEmail();

    public void setEdat(String edat);

    public void setMessage(String message);

    public void setEmail(String email);

}
```

Fixeu-vos que el mètode `setEdat` rep un *string* com a paràmetre i el `getEdat` retorna un *integer*.

Quan s'utilitza el mètode `setEdat`, la informació, originàriament, es troba en un formulari. El protocol HTTP envia les dades al *servlet* i aquest cridarà el mètode `setEdat`. Totes les dades que s'envien amb un formulari són de tipus *string*. Així, la conversió de les dades es farà en el mateix *bean*.

El codi amb la implementació del *bean* és el següent:

```
@Stateful
public class PostBean2 implements PostBean2Local {

    private int edat;

    private String message;

    private String email;

    @Override
    public int getEdat() {
        return edat;
    }

    @Override
    public void setEdat(String edat) {
         if(!edat.equals("")){
            this.edat=Integer.parseInt(edat);
        }
        else{
            this.edat = 0;
        }
    }

    @Override
    public String getMessage() {
        return message;
    }
```

```
29
30      @Override
31      public void setMessage(String message) {
32          this.message = message;
33      }
34
35      @Override
36      public String getEmail() {
37          return email;
38      }
39
40      @Override
41      public void setEmail(String email) {
42          this.email = email;
43      }
44  }
```

Sembla un *bean* normal. L'única diferència és l'anotació `@Stateful` en declarar la classe del *bean*. Aquesta notació informa el servidor que es tracta d'un *bean* EJB amb sessió i estat.

Conté les propietats *edat*, *missatge* i *correu electrònic*. A més a més, implementa la interfície PostBean2Local.

Ara bé, on es posen les validacions? Hem dit que s'utilitzaran les anotacions del paquet Java Beans Validation, i aquestes es poden posar a les propietats, als mètodes o a la mateixa classe. On les poseu?

Heu de tenir en compte, per respondre a aquesta pregunta, que l'aplicació que utilitzi el *bean* no podrà accedir-hi directament. Llavors, si voleu veure que les restriccions de seguretat s'estan aplicant, necessiteu utilitzar la classe a la qual podeu accedir directament.

Aquesta classe és la interfície del *bean*. Llavors, com que aquesta interfície només posseeix els mètodes del *bean*, i no les seves propietats, posareu les validacions en els mètodes.

Fixeu-vos en la nova implementació d'aquesta interfície:

```
1   @Local
2   public interface PostBean2Local {
3
4       @Min(value=18, message = "Has de ser major d'edat per escriure un missatge.
            ")
5       public int getEdat();
6
7       @NotNull @Size(min=1, max=150, message = "El missatge no és vàlid. Ha de
            tenir menys de 150 caràcters.")
8       public String getMessage();
9
10      @NotNull @Pattern(regexp="^(.+)@(.+)$", message = "El correu no és vàlid.")
11      public String getEmail();
12
13      public void setEdat(String edat);
14
15      public void setMessage(String message);
16
17      public void setEmail(String email);
18
19  }
```

Per fer les validacions de cadascuna de les dades s'han utilitzat les següents anotacions:

- `@Min`: comprova que el valor sigui un enter i sigui menor que l'atribut `value` passat com a paràmetre de la restricció. En el cas que es compleixi, es genera un violació de la restricció (`ConstraintViolation`) que té com a missatge el text de l'atribut `message`.
- `@NotNull`: comprova si el valor és *null*. Si és *null* es genera una violació de la restricció.
- `@Size`: admet dos atributs: `min` i `max`. La mida del text ha d'estar entre aquests dos valors. Si no és això es genera una violació de la restricció que té com a missatge el text de l'atribut `message`.
- `@Pattern`: admet una expressió regular. La dada que retorna la funció ha de satisfer l'expressió regular. En el cas que no la compleixi es mostrarà per pantalla el missatge de l'atribut `message`.

Heu posat les restriccions en els mètodes `get` perquè us interessa que sigui quan vulguem accedir al valor que ens indiqui si es compleix la restricció.

Totes aquestes restriccions estan creades per defecte i pertanyen al paquet Java Validation. En aquest cas no heu necessitat crear restriccions noves.

Una vegada hem acabat d'implementar les validacions i el *bean* amb estat, creareu el *servlet* que rebrà les dades del formulari HTML i les enviarà al bean PostBean2. Creareu un nou *servlet* ( *File / New File / Web / Servlet* ) i l'anomenareu PostServlet2. No activareu la utilització del fitxer web.xml durant la seva creació, ja que fareu servir anotacions.

La implementació del *servlet* PostServlet2 és la següent:

```
1  @WebServlet(name = "PostServlet2", urlPatterns = {"/PostServlet2"})
2  public class PostServlet2 extends HttpServlet {
3
4      @Resource Validator validator;
5
6      protected void processRequest(HttpServletRequest request,
           HttpServletResponse response)
7              throws ServletException, IOException {
8
9              response.setContentType("text/html;charset=UTF-8");
10             try (PrintWriter out = response.getWriter()) {
11             out.println("<!DOCTYPE html>");
12             out.println("<html>");
13             out.println("<head>");
14             out.println("<title>Servlet ServletValidation2</title>");
15             out.println("</head>");
16             out.println("<body>");
17
18             String missatge = request.getParameter("missatgePost");
19             String mail = request.getParameter("mail");
20             String edat = request.getParameter("edat");
21
22             PostBean2Local bean = (PostBean2Local) new InitialContext().lookup(
23                     "java:global/formServlets/PostBean2");
24
25
```

```
            bean.setMessage(missatge);
            bean.setEmail(mail);
            bean.setEdat(edat);

            out.println("<h1>Dades rebudes del formulari</h1>");
            out.println("<p>Missatge: " + bean.getMessage() + "</p>");
            out.println("<p>Edat: " + bean.getEdat() + "</p>");
            out.println("<p>Email: " + bean.getEmail() + "</p>");

            out.println("<h1>Llistat de validacions:</h1>");
            for (ConstraintViolation c : validator.validate(bean)) {
                out.println("<p>" + c.getMessage() + "</p>");
            }

            out.println("</body>");
            out.println("</html>");
            } catch (NamingException ex) {
            Logger.getLogger(PostServlet2.class.getName()).log(Level.SEVERE,
                null, ex);
        }
    }
```

Bàsicament, el *servlet* rep les dades enviades per l'usuari:

```
String missatge = request.getParameter("missatgePost");
String mail = request.getParameter("mail");
String edat = request.getParameter("edat");
```

i les introdueix en el *bean*:

```
PostBean2Local bean = (PostBean2Local) new InitialContext().lookup(
                    "java:global/formServlets/PostBean2");

bean.setMessage(missatge);
bean.setEmail(mail);
bean.setEdat(edat);
```

Com que és un *bean* amb estat, s'ha de demanar la seva utilització. L'objecte `InitialContext` és l'encarregat de cercar qualsevol tipus d'objecte. Per tal que pugui trobar-lo, li heu de dir el seu nom i el projecte al qual pertany.

```
PostBean2Local bean = (PostBean2Local) new InitialContext().lookup(
                    "java:global/formServlets/PostBean2");
```

El codi anterior no crea l'objecte, us en cerca un per a vosaltres. Fixeu-vos que quan demanem per l'objecte `PostBean2` ens retornen un objecte del tipus `PostBean2Local`. Això és degut al fet que el contenidor EJB no permet accedir directament a l'objecte.

Una vegada teniu l'objecte, ja podeu introduir les dades de l'usuari:

```
bean.setMessage(missatge);
bean.setEmail(mail);
bean.setEdat(edat);
```

Una vegada el *bean* té les dades es força la seva validació manualment. Si hi ha alguna violació de les restriccions posades en els mètodes, en validar-se es generen les `ConstraintViolation`. El missatge d'aquestes violacions de restricció es mostrarà per pantalla.

```
for (ConstraintViolation c : validator.validate(bean)) {
  out.println("<p>" + c.getMessage() + "</p>");
}
```

En el cicle de vida dels *servlets* no existeix una etapa de validació de les dades. Aquesta s'ha de fer manualment, per això necessiteu accedir al validador de Java Validation. En canvi, en un *framework* com pot ser Spring o JSF, el validador s'executa automàticament en un moment determinat del cicle de vida del *framework*.

Per accedir al validador s'ha creat una propietat del *servlet* del tipus `Validator` i s'ha informat que aquest és un recurs de Java. El codi és el següent:

```
@Resource Validator validator;
```

L'etiqueta `@Resource` informa el servidor que l'aplicació necessita un validador. Aquest validador s'injecta, igual que el contenidor EJB injecta *beans* sense estat, la primera vegada que s'utilitza.

Una vegada teniu el validador, s'executa el mètode `validar` i per paràmetre s'envia el *bean* que es vol validar.

```
validator.validate(bean)
```

Aquest validador utilitzarà els mètodes `get` per obtenir les dades, i si hi ha alguna violació de restricció s'emmagatzemarà per retornar-lo una vegada s'hagin completat totes les validacions.

Es pot iterar aquest conjunt per accedir a totes les violacions i mostrar-les per pantalla:

```
for (ConstraintViolation c : validator.validate(bean)) {
  out.println("<p>" + c.getMessage() + "</p>");
}
```

Una vegada s'ha acabat d'implementar el *servlet*, creareu una rèplica de la pàgina HTML de l'exercici anterior on demanarem l'edat, el correu electrònic i el missatge per mostrar per pantalla. L'únic canvi serà el recurs al qual se li enviaran les dades (el mètode `action` del formulari). A aquesta pàgina l'anomenareu escriurePost2.html. El contingut de la pàgina és el següent:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Bean Validation</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
      <h1>Escriu un missatge:</h1>
        <form id="formulari" method="POST" action="PostServlet2">
            <label for="mail">Correu electrònic:</label>
            <input id="mail" type="text" name="mail"/>
            <label for="age">Edat:</label>
            <input id="age" type="number" name="edat" min="0">
            <input type="submit">
```

```
16          </form>
17          <label>Missatge:</label>
18          <textarea name="missatgePost" form="formulari"></textarea>
19      </body>
20  </html>
```

Vegeu el funcionament de l'aplicació Escriure un Post2. A la figura 3.9 podeu veure com ens mostra un formulari per introduir-hi les dades: edat, correu electrònic i el missatge.

**FIGURA 2.9.** Pantalla inicial de l'aplicació Escriure un Post2


A la següent pantalla, les dades es validen amb EJB i es mostra el resultat de la validació, com podeu veure en la figura 3.10.

**FIGURA 2.10.** Pantalla de validació de l'aplicació Escriure un Post2


## 2.3 Dades de subscripció

En aquest apartat s'aprofundeix en la validació de dades utilitzant EJB de sessió amb estat. Anteriorment s'ha fet la validació amb restriccions proporcionades per la llibreria Java Validation (*built-in constraint*), i en aquest apartat introduirem les validacions de restriccions d'usuari (*custom constraint*).

Es vol crear un registre d'usuaris de l'aplicació. Es demanarà a l'usuari la següent informació:

- nom

- cognoms
- data de naixement
- gènere
- correu electrònic
- telèfon
- color favorit
- marca de cotxe
- vehicle1 (valor “bicicleta”, si l’usuari en posseeix una)
- vehicle2 (valor “moto”, si l’usuari en posseeix una)
- navegador d’Internet favorit

S’han de validar totes les dades. En concret, es vol comprovar que:

- No siguin *null* els camps nom, cognoms, gènere, marca de cotxe i navegador d’Internet favorit.
- No estiguin buits els camps nom, cognom, data de naixement, correu electrònic, telèfon, vehicle1, vehicle2 i navegador d’Internet.
- L’usuari sigui major d’edat.
- El correu electrònic i el telèfon siguin vàlids.
- El color estigui en hexadecimal i tingui la forma ”#hhhhhh” o ”#hhh”.

**Java Interface**

Una interfície a Java és una classe abstracta que especifica els mètodes que han d’implementar totes les classes que implementin aquesta interfície. Les classes que implementin una interfície han d’implementar tots els mètodes o declarar-se també com a abstractes. L’avantatge d’utilitzar interfícies és que simulen l’herència múltiple.

El projecte utilitzat serà el mateix que heu emprat fins ara, i començareu creant el *bean* EJB amb estat. Creareu també, automàticament, la interfície local associada. Per crear el *bean* amb l’IDE NetBeans aneu a *File / New File / Enterprise JavaBeans / Session Bean* i l’anomeneu SignupBean.

Aquest *bean* serà un *bean* de sessió amb estat; per tant, haurà d’aparèixer l’anotació `@Stateful` abans de la declaració de la classe. A més a més, tindrà totes les propietats corresponents al registre de l’usuari. Podeu veure la seva implementació en el següent codi:

```
1  @Stateful
2  public class SignupBean implements SignupBeanLocal {
3
4      private String nom;
5
6      private String cognoms;
7
8      private String naixement;
9
10     private String sexe;
11
12     private String email;
13
14     private String telefon;
```

```
15
16      private String color;
17
18      private String marcaCotxe;
19
20      private String vehicle1;
21
22      private String vehicle2;
23
24      private String navegador;
25
26      @Override
27      public String print() {
28
29          return "<p>nom=" + nom + "</p> <p>"
30                  + "cognoms=" + cognoms + "</p><p>"
31                  + "naixement=" + naixement + "</p><p>"
32                  + "sexe=" + sexe + "</p> <p>email=" + email + "</p><p>"
33                  + "telefon=" + telefon + "</p> <p>"
34                  + "color=" + color + "</p> <p>"
35                  + "marcaCotxe=" + marcaCotxe + "</p> <p>"
36                  + "vehicle1=" + vehicle1 + "</p> <p>"
37                  + "vehicle2=" + vehicle2 + "</p> <p>"
38                  + "navegador=" + navegador + "</p>";
39      }
40
41      @Override
42      public String getNom() {
43          return nom;
44      }
45
46      @Override
47      public void setNom(String nom) {
48          this.nom = nom;
49      }
50
51     @Override
52      public String getCognoms() {
53          return cognoms;
54      }
55
56      @Override
57      public void setCognoms(String cognoms) {
58          this.cognoms = cognoms;
59      }
60
61      @Override
62      public String getNaixement() {
63          return naixement;
64      }
65
66      @Override
67      public void setNaixement(String naixement) {
68          this.naixement = naixement;
69      }
70
71      @Override
72      public String getSexe() {
73          return sexe;
74      }
75
76      @Override
77      public void setSexe(String sexe) {
78          this.sexe = sexe;
79      }
80
81      @Override
82      public String getEmail() {
83          return email;
84      }
```

```
85
86      @Override
87      public void setEmail(String email) {
88          this.email = email;
89      }
90
91      @Override
92      public String getTelefon() {
93          return telefon;
94      }
95
96      @Override
97      public void setTelefon(String telefon) {
98          this.telefon = telefon;
99      }
100
101     @Override
102     public String getColor() {
103         return color;
104     }
105
106     @Override
107     public void setColor(String color) {
108         this.color = color;
109     }
110
111     @Override
112     public String getMarcaCotxe() {
113         return marcaCotxe;
114     }
115
116     @Override
117     public void setMarcaCotxe(String marcaCotxe) {
118         this.marcaCotxe = marcaCotxe;
119     }
120
121     @Override
122     public String getVehicle1() {
123         return vehicle1;
124     }
125
126     @Override
127     public void setVehicle1(String vehicle1) {
128         this.vehicle1 = vehicle1;
129     }
130
131     @Override
132     public String getVehicle2() {
133         return vehicle2;
134     }
135
136     @Override
137     public void setVehicle2(String vehicle2) {
138         this.vehicle2 = vehicle2;
139     }
140
141     @Override
142     public String getNavegador() {
143         return navegador;
144     }
145
146     @Override
147     public void setNavegador(String navegador) {
148         this.navegador = navegador;
149     }
150
151 }
```

De moment, la interfície local associada al *bean* anterior només conté els *getters* i *setters* de les propietats del *bean*.

A continuació s'han d'introduir les restriccions de validació necessàries per assegurar-nos que les dades compleixen els requisits establerts.

Les llibreries de validació proporcionen les restriccions més habituals. Així, podeu utilitzar les següents anotacions estàndard:

- `@NotNull`: s'aplicarà als mètodes de la interfície local SignupBeanLocal que es vulgui comprovar que no és *null*. Llavors, aquesta anotació s'aplicarà als mètodes `get` de les propietats: nom, cognoms, gènere, marca de cotxe i navegador d'Internet favorit.
- `@Pattern`: aquesta anotació només s'aplicarà per comprovar que el telèfon introduït és vàlid. Creareu una expressió regular que ho comprovi.

Vegeu com aplicar les restriccions anteriors a la interfície local:

```
@Local
public interface SignupBeanLocal {

    @NotNull (message = "El nom no pot estar buit.")
    public String getNom();

    @NotNull(message = "El cognom no pot estar buit.")
    public String getCognoms();

    public String getNaixement();

    @NotNull(message = "El gènere no pot estar buit.")
    public String getSexe();

    public String getEmail();

    @Pattern(regexp = "\\d{3}\\-\\d{3}-\\d{3}", message = "El telèfon no és vàlid.")
    public String getTelefon();

    public String getColor();

    @NotNull(message = "La marca del cotxe no pot estar buida.")
    public String getMarcaCotxe();

    public String getVehicle1();

    public String getVehicle2();

    @NotNull(message = "El navegador no pot estar buit.")
    public String getNavegador();

    public String print();

    public void setNom(String nom);

    public void setCognoms(String cognoms);

    public void setNaixement(String naixement);

    public void setSexe(String sexe);

    public void setEmail(String email);

    public void setTelefon(String telefon);

    public void setColor(String color);

```

```
48      public void setMarcaCotxe(String marcaCotxe);
49
50      public void setVehicle1(String vehicle1);
51
52      public void setVehicle2(String vehicle2);
53
54      public void setNavegador(String navegador);
55  }
```

Observeu que només s'ha de modificar el missatge de la restricció `@NotNull` perquè funcioni de la manera esperada. En canvi, a la restricció `@Pattern` de la propietat "telèfon" s'ha d'afegir l'expressió regular que utilitzarà per comprovar si és vàlid.

L'expressió utilitzada és la següent:

Amb la sintaxi *{nombre}*, d'una expressió regular, s'indica el nombre de repeticions de l'expressió anterior. Llavors, quan s'indica *\d{3}* significa que es volen 3 dígits.

```
1  @Pattern(regexp = "\\d{3}\\-\\d{3}-\\d{3}", message = "El telèfon no és vàlid."
       )
```

La resta de restriccions no es troben implementades per defecte a la llibreria de validacions. Heu de crear les restriccions d'usuari següents per fer comprovacions específiques de la vostra aplicació. El fet de crear aquestes restriccions permet reutilitzar-les posteriorment en altres punts del programa:

- `@Check18`: comprova que l'usuari sigui major d'edat. Rep un *string* per paràmetre i comprova que a data d'avui sigui més gran de 18 anys.

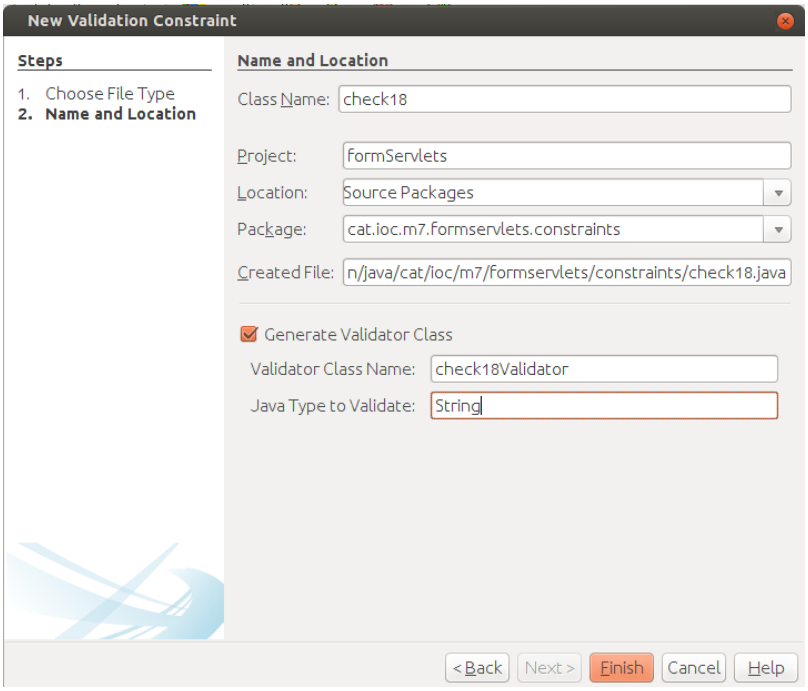
- `@Color`: utilitzant una expressió regular comprova que el color introduït tingui el format *#cccccc* o bé *#ccc*, on *c* és un nombre hexadecimal. No s'utilitza la restricció *@Pattern* perquè es vol reaprofitar posteriorment.

- `@Email`: utilitzant una expressió regular comprova que l'adreça electrònica sigui vàlida. No s'utilitza la restricció *@Pattern* perquè es vol reaprofitar posteriorment.

- `@NotBlank`: comprova que el text no sigui buit (””).

Començareu creant la restricció `@Check18`. Anireu a *File / New File / Bean Validation / Validation Constraint*, l'anomenareu Check18 i activareu la creació d'una classe validadora del tipus *string* (vegeu la figura 3.11).

**FIGURA 2.11.** Creació d'una restricció d'usuari nova




Aquest procediment crea dues classes. La classe `Check18.java` és la descripció de l'anotació `@Check18`. S'especifiquen els paràmetres de la restricció tals com si és una restricció de mètode, de classe o d'atribut.

```
@Documented
@Constraint(validatedBy = Check18Validator.class)
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Check18 {

    String message() default "{cat.ioc.m7.formservlets.constraints.PastDate}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

El més important de la classe anterior és el validador que s'utilitza. Observeu que amb l'anotació `@Constraint` s'informa que la classe `Check18Validator` és el validador associat a aquesta anotació. És a dir, allà on s'apliqui aquesta anotació, qui realment comprova si la dada és vàlida serà aquesta classe.

L'anotació `@Target` especifica on es pot utilitzar l'anotació `@Check18`. En aquest cas, s'ha permès la seva utilització en la declaració de mètodes, de propietats i d'altres anotacions.

L'altre classe que es crea és el validador de la classe. En aquest cas, el seu nom és `Check18Validator`. Aquesta classe ha de fer la validació de la dada. Per això, en la seva creació heu d'informar del tipus de dada que validarà.

Vegeu la implementació d'aquesta classe:

```
public class Check18Validator implements ConstraintValidator<Check18, String> {

    @Override
```

```
    public void initialize(Check18 constraintAnnotation) {
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        try {
            if (!value.equals("")) {
                Date naixement = new SimpleDateFormat("yyyy-MM-dd").parse(value
                    );
                int anyAra = Calendar.getInstance().get(Calendar.YEAR);
                Calendar cal = Calendar.getInstance();
                cal.setTime(naixement);
                int anyNaixement = cal.get(Calendar.YEAR);
                return anyAra - anyNaixement >= 18;
            }
        } catch (ParseException ex) {
            Logger.getLogger(Check18Validator.class.getName()).log(Level.SEVERE
                , null, ex);
        }
        return false;
    }
}
```

En crear el validador, NetBeans afegeix dues excepcions: una en el mètode `initialize` i l'altra en el mètode `isValid`. Recordeu de treure-les perquè funcioni la validació.

El mètode `Initialize` s'executarà sempre abans del mètode `isValid`, i s'utilitza per inicialitzar qualsevol propietat que necessiti el validador.

El mètode `isValid` és qui fa la validació de la dada. Retorna *false* si la restricció es compleix i, per tant, s'ha de generar una violació de restricció. Retorna *true* en cas contrari.

El paràmetre *value* del mètode `isValid` correspon a la dada de la propietat a validar, i no es pot sobreescriure.

En aquest cas s'ha de calcular l'edat de la persona. Observeu la seva implementació:

```
if (!value.equals("")) {
   Date naixement = new SimpleDateFormat("yyyy-MM-dd").parse(value);
   int anyAra = Calendar.getInstance().get(Calendar.YEAR);
   Calendar cal = Calendar.getInstance();
   cal.setTime(naixement);
   int anyNaixement = cal.get(Calendar.YEAR);
   return anyAra - anyNaixement >= 18;
}
return false;
```

Si el valor `value` no és buit es transforma en tipus `Date`. S'aconsegueix l'any actual utilitzant l'objecte `Calendar` i després, també amb el mateix objecte, s'aconsegueix l'any de la data de naixement. Si la resta, entre l'any actual i l'any del dia que va néixer, és més petita que 18, es retorna *false* i, llavors, es genera una violació de restricció. Es retorna *true* en cas contrari i no es genera cap violació de restricció.

Una vegada s'ha implementat aquest mètode ja es pot utilitzar. Vegeu com s'aplica al mètode `getNaixement` de la interfície SignupBeanLocal:

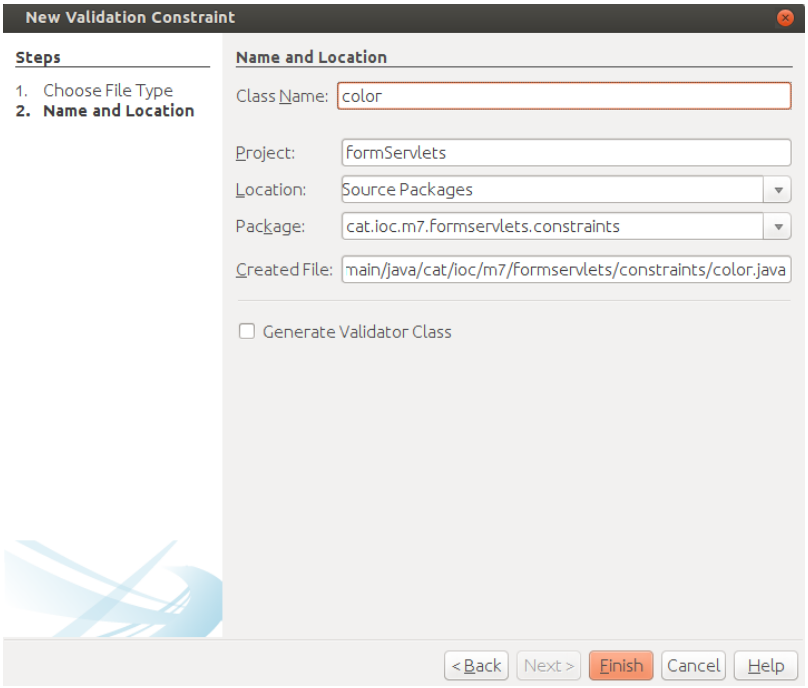
```
@Check18 (message = "Has de tenir més de 18 anys.")
public String getNaixement();
```

Existeix una altra manera per crear restriccions. La idea és que, a vegades, no cal crear un validador nou, perquè es pot aprofitar una altra restricció per validar la que es vol crear.

Per exemple, voleu crear una validació pel color. Com que per validar aquesta restricció utilitzareu una expressió regular i existeix ja una restricció per a *patterns*, llavors no cal crear un validador nou. Reaprofitareu aquesta restricció (`@Pattern`) i aplicareu l'expressió regular concreta.

Començareu creant una nova restricció (*File / New File / Bean Validation / Validation Constraint* ) anomenada Color.java sense activar la creació d'un validador (vegeu la figura 3.12).

**FIGURA 2.12.** Creació d'una restricció sense validador




Una vegada creada la restricció *Color* afegireu l'anotació `@Pattern` com a anotació de restricció de la interfície. Aquesta anotació `@Pattern` accepta un paràmetre corresponent al llistat de *patterns* a comprovar. Vegeu-ho:

```
@Pattern.List({@Pattern(regexp = "#([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})")})
```

En incloure aquest llistat, la classe `Color` ha de definir una interfície anomenada List amb mètode anomenat `value()` que retorni un *array* de la classe `Color`. Exemple:

```
@interface List {
   Color[] value();
}
```

En introduir aquests dos elements ja no és necessari disposar d'un validador específic per a la classe `Color`, ja que s'utilitzarà la restricció `@Pattern` amb l'expressió regular "#([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})".

Finalment, la restricció Color.java està implementada de la següent manera:

```
@Pattern.List({@Pattern(regexp = "#([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})")})
@Constraint(validatedBy = {})
@Documented
@Target({ElementType.METHOD,
    ElementType.FIELD,
    ElementType.ANNOTATION_TYPE,
    ElementType.CONSTRUCTOR,
    ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Color {

    String message() default "{invalid.color}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ElementType.METHOD,
        ElementType.FIELD,
        ElementType.ANNOTATION_TYPE,
        ElementType.CONSTRUCTOR,
        ElementType.PARAMETER})
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    @interface List {
        Color[] value();
    }
}
```

Observeu com l'anotació `@Constraint`, que s'utilitza per indicar el validador a fer servir, està buida.

Ara ja es pot aplicar aquesta restricció al mètode `getColor` de la classe `SignUpBeanLocal`.

```
@Color(message = "El color no pot estar buit.")
public String getColor();
```

Falta crear dues restriccions més: la restricció anomenada `@NotBlank`, que comprova si una cadena de caràcters és buida (""), i la restricció anomenada `@Email`, que comprova si un *mail* és vàlid amb una expressió regular.

Es proposa que intenteu fer vosaltres la implementació d'aquestes dues restriccions. La primera restricció, `@NotBlank`, s'ha de crear amb un validador associat i, en canvi, la restricció `@Email` cal crear-la sense validador associat utilitzant la restricció `@Pattern`.

Podeu trobar la solució d'aquestes restriccions en el codi proporcionat amb aquest apartat.

Finalment, la classe `SignUpBeanLocal` ja disposa de totes les validacions demanades:

```
@Local
```

```
public interface SignupBeanLocal {

    @NotNull (message = "El nom no pot estar buit.")
    @NotBlank(message = "El nom no pot estar buit.")
    public String getNom();

    @NotNull(message = "El cognom no pot estar buit.")
    @NotBlank(message = "El cognom no pot estar buit.")
    public String getCognoms();

    @Check18 (message = "Has de tenir més de 18 anys.")
    @NotBlank(message = "La data de naixement no pot estar buida.")
    public String getNaixement();

    @NotNull(message = "El gènere no pot estar buit.")
    public String getSexe();

    @Email
    @NotBlank(message = "El correu no pot estar buit.")
    public String getEmail();

    @Pattern(regexp = "\\(\\d{3}\\)\\d{3}-\\d{4}", message = "El telèfon no és
        vàlid.")
    @NotBlank(message = "El telèfon no pot estar buit.")
    public String getTelefon();

    @Color(message = "El color no pot estar buit.")
    public String getColor();

    @NotNull(message = "La marca del cotxe no pot estar buida.")
    public String getMarcaCotxe();

    @NotBlank(message = "El vehicle1 no pot estar buit.")
    public String getVehicle1();

    @NotBlank(message = "El vehicle2 no pot estar buit.")
    public String getVehicle2();

    @NotNull(message = "El navegador no pot estar buit.")
    @NotBlank(message = "El navegador no pot estar buit.")
    public String getNavegador();

    public String print();

    public void setNom(String nom);

    public void setCognoms(String cognoms);

    public void setNaixement(String naixement);

    public void setSexe(String sexe);

    public void setEmail(String email);

    public void setTelefon(String telefon);

    public void setColor(String color);

    public void setMarcaCotxe(String marcaCotxe);

    public void setVehicle1(String vehicle1);

    public void setVehicle2(String vehicle2);

    public void setNavegador(String navegador);
}
```

Una vegada heu acabat d'implementar les validacions i el *bean* amb estat, creareu el *servlet* que rebrà les dades del formulari HTML i les enviarà al *bean* Signup-

Bean. Creareu un nou *servlet* (*File / New File / Web / Servlet*) i l'anomenareu SignUpServlet. No activareu la utilització del fitxer web.xml durant la seva creació, ja que fareu servir anotacions.

La implementació del *servlet* SignUpServlet és la següent:

```
@WebServlet(name = "SignUpServlet", urlPatterns = {"/SignUpServlet"})
public class SignUpServlet extends HttpServlet {


    @Resource Validator validator;

    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet SingUp</title>");
            out.println("</head>");
            out.println("<body>");

            SignupBeanLocal bean = (SignupBeanLocal) new InitialContext().
                lookup(
                    "java:global/formServlets/SignupBean");

            try{
                out.println("<h1>Dades rebudes del formulari</h1>");
                BeanUtils.populate (bean, request.getParameterMap());
                out.println("<p>" + bean.print() + "</p>");
            }
            catch(IllegalAccessException | InvocationTargetException e){
                out.println(e.getMessage());
            }

            out.println("<h1>Llistat de validacions:</h1>");

            for (ConstraintViolation c : validator.validate(bean)) {
                out.println("<p>" + c.getMessage() + "</p>");
            }

            out.println("</body>");
            out.println("</html>");
        } catch (NamingException ex) {
            Logger.getLogger(SignUpServlet.class.getName()).log(Level.SEVERE,
                null, ex);
        }
    }
```

La implementació del *servlet* és molt semblant al *servlet* PostServlet2. Per no ser repetitius només s'explicaran les diferències.

La primera diferència és el *bean* que es recupera del contenidor EJB mitjançant l'objecte `InitialContext`.

```
SignupBeanLocal bean = (SignupBeanLocal) new InitialContext().lookup(
                    "java:global/formServlets/SignupBean");
```

L'objecte EJB a recuperar és `SignupBean`, però el contenidor EJB retorna un objecte del tipus `SignupBeanLocal`, tal com s'esperava.

La segona diferència és la utilització de la classe `BeanUtils` per associar les propietats del *bean* amb les dades enviades per l'usuari. En utilitzar aquesta classe s'ha de cercar la seva dependència amb el programa NetBeans.

**Apache Commons BeanUtils**

La biblioteca de Java proporciona mecanismes per accedir dinàmicament a les propietats d'un objecte, és a dir, sense utilitzar els seus mètodes *getXXX* o *setXXX*. Aquests mecanismes són bastant complexos i requereixen l'ús dels paquets *java.lang.reflect* i *java.beans*. El component BeanUtils d'Apache proporciona un mecanisme fàcil d'utilitzar per accedir a aquestes funcionalitats.

```
BeanUtils.populate (bean, request.getParameterMap());
```

Perquè funcioni el mètode `populate` de l'objecte `BeanUtils` s'ha de crear el formulari web de tal manera que el nom dels *inputs* del formulari correspongui amb el nom dels mètodes *set* de les propietats. Exemple:

```
<input id="naix" type="date" name="naixement"><br>
```

L'atribut *name* de l'*input* és *naixement*, i aquest correspon al nom de la propietat del *bean*:

```
private String naixement;
```

Aquesta propietat genera un mètode `setNaixement` que és cridat pel mètode `populate` de l'objecte `BeanUtils` en trobar un paràmetre anomenat *naixement* en l'objecte `request`.

D'aquesta manera us estalvieu de fer manualment tots els *setters* dels paràmetres.

Una vegada heu acabat d'implementar el *servlet*, creareu la pàgina HTML associada amb el formulari web i l'anomenareu subscripcio.html.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Subscripció</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <h1>Per subscriure't necessitem les següents dades:</h1>
        <form id="formulari" method="POST" action="SignUpServlet">
            <label for="nom">Nom:</label>
            <input type="text" name="nom" id="nom"><br>
            <label for="cognoms">Cognoms:</label>
            <input type="text" name="cognoms" id="cognoms"><br>
            <label for="naix">Data de naixement:</label>
            <input id="naix" type="date" name="naixement"><br>
            <input type="radio" name="sexe" value="home" checked> Home<br>
            <input type="radio" name="sexe" value="dona"> Dona<br>
            <label for="email">Correu electrònic:</label>
            <input id="email" type="text" name="email"><br>
            <label for="telf">Telèfon:</label>
            <input id="telf" type="tel" name="telefon"><br>
            <label for="color">Color preferit:</label>
            <input id="color" type="color" name="color"><br>

            <label for="cotxe">Marca del cotxe:</label>
            <select id="cotxe" name="marcaCotxe">
                <option value="volvo">Volvo</option>
                <option value="saab">Saab</option>
                <option value="fiat">Fiat</option>
                <option value="audi">Audi</option>
                <option value="audi">Altre</option>
            </select><br>
            <input type="checkbox" name="vehicle1" value="Bicicleta"> Tinc una
                bicicleta<br>
```

```
35              <input type="checkbox" name="vehicle2" value="Moto"> Tinc una moto<
                    br>
36              <input list="browsers" name="navegador">
37              <datalist id="browsers">
38                  <option value="Internet Explorer">
39                  <option value="Firefox">
40                  <option value="Chrome">
41                  <option value="Opera">
42                  <option value="Safari">
43              </datalist>
44              <input type="submit">
45          </form>
46
47      </body>
48  </html>
```

Vegeu el funcionament de l'aplicació Subscripció. En la figura 3.13 podeu veure com ens mostra un formulari per introduir-hi les dades.

**FIGURA 2.13.** Pantalla inicial de l'aplicació subscripció


Finalment, les dades es validen amb EJB i es mostra el resultat de la validació, com podeu veure en la figura 3.14:

**FIGURA 2.14.** Resultat de la validació de la subscripció

## 2.4 Què s'ha après?

En finalitzar aquest apartat l'estudiant ha après a:

- Tractar les dades enviades mitjançant un formulari a un *servlet*
- Utilitzar l'arquitectura EJB juntament amb l'arquitectura Java Web
- Diferenciar entre un Java Bean i un Enterprise Java Bean
- Diferenciar entre un Bean EJB amb estat i un sense estat
- Crear i utilitzar expressions regulars
- Crear i utilitzar les restriccions (*constraints*) EJB

En acabar aquesta apartat, els estudiants ja estan preparats per començar les activitats proposades en aquest apartat i poden continuar amb el seu aprenentatge del llenguatge Java Web.

## 3. Manteniment d'estat, autenticació i autorització amb 'servlets' i EJB

En aquest apartat s'explicaran les diferents maneres que té el programador Java Web per mantenir l'estat de l'aplicació. Moltes vegades ens interessa reconèixer un usuari que ha entrat anteriorment al programa i ha seleccionat les seves preferències a l'hora d'interactuar amb l'aplicació.

Existeixen diferents maneres d'emmagatzemar aquest tipus de dades, des de tècniques molt avançades fins a algunes que us podeu inventar per tal de guardar la informació on volem. Però, en general, existeixen dues maneres d'aconseguir-ho.

La primera manera consisteix en el fet que la informació l'ha d'emmagatzemar el mateix usuari. Així, cada vegada que es comuniqui amb el servidor automàticament enviarà tota la informació que s'havia emmagatzemat durant l'última visita al programa. Les diferents tècniques que s'utilitzen per emmagatzemar la informació en el costat client corresponen a la utilització de galetes, formularis amb camps ocults i reescriptura d'adreces web (URL) amb paràmetres.

La segona manera consisteix a guardar la informació en el costat del servidor. El client només s'ha d'identificar per tal d'obtenir les dades que s'havien emmagatzemat. La tècnica que consisteix a guardar la informació en el costat del servidor s'anomena *sessió*. És una tècnica molt útil que, junt amb les tècniques anteriors, es pot utilitzar per fer aplicacions interactives i amb una experiència molt satisfactòria per part de l'usuari.

A més a més, també s'explicarà com autenticar un usuari que està intentant accedir a un recurs del sistema. Normalment, l'autenticació del client es fa mitjançant un usuari i una contrasenya. Una vegada autenticat, depenent del rol de l'usuari, aquest podrà accedir a unes funcions o a unes altres.

Per a aquesta última part de la unitat es recomana haver realitzat l'apartat anomenat "Formularis amb *servlets* i EJB", on s'expliquen els diferent elements que intervenen en l'arquitectura EJB.

Podeu descarregar-vos el codi Java utilitzat en aquest apartat des de l'enllaç que trobareu a l'apartat d'annexos de la unitat. Recordeu que podeu utilitzar la funció d'importar del Netbeans per accedir a aquest contingut.

### 3.1 L'usuari, en una galeta

En aquest apartat aprendrem a emmagatzemar informació utilitzant galetes. El navegador guarda la informació en forma d'arxiu de text al disc dur del visitant de la pàgina web per tal que certes informacions puguin ser recuperades en posteriors visites.

Comencem creant un nou projecte amb Maven, i posteriorment crearem un formulari en una pàgina web que enviï les dades d'un usuari a un *servlet* per tal que

l'emmagatzemi en una galeta (*cookie*). La pàgina web l'anomenarem galetes.html, i el seu contingut és el següent:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Galetes</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>Emmagatzemar l'usuari a una galeta.</h1>
        <h2>Introdueix el teu nom:</h2>
        <form action="galetesServlet" method="post">
            <label for="nom"> Nom:</label>
            <input id="nom" type="text" name="userName"/>  <br/>
            <input type="submit" value="enviar"/>
        </form>
    </body>
</html>
```

Com podeu veure a la figura 3.1, la pàgina web envia el nom d'usuari al *servlet* anomenat galetesServlet. Heu de crear aquest *servlet*, que rebrà el nom d'usuari i l'emmagatzemarà en una galeta.

**FIGURA 3.1.** Enviar dades al 'servlet' per crear una galeta


Recordeu que per crear el *servlet* amb NetBeans heu d'accedir a *File / New File / Web / Servlet* . No cal que afegiu la configuració del *servlet* al fitxer web.xml. En aquest cas, utilitzarem les anotacions per configurar-lo. El seu nom serà galetesServlet, i la implementació del seu codi és la següent:

```
@WebServlet(name = "galetesServlet", urlPatterns = {"/galetesServlet"})
public class GaletesServlet extends HttpServlet {

    /**
     * Processes requests for both HTTP GET and POST
     * methods.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
```

```
14              throws ServletException, IOException {
15          response.setContentType("text/html;charset=UTF-8");
16          try (PrintWriter out = response.getWriter()) {
17
18              out.println("<!DOCTYPE html>");
19              out.println("<html>");
20              out.println("<head>");
21              out.println("<title>Servlet galetes1</title>");
22              out.println("</head>");
23              out.println("<body>");
24              out.println("<h1>Ara guardem la galeta en el teu navegador</h1>");
25
26              String n = request.getParameter("userName");
27              out.print("Benvingut " + n);
28
29              Cookie ck = new Cookie("usuari", n);
30              response.addCookie(ck);
31
32              out.print("<form action='galetesServlet2'>");
33              out.print("<input type='submit' value='anar'>");
34              out.print("</form>");
35
36              out.println("</body>");
37              out.println("</html>");
38          }
39      }
40
41      // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click
            on the + sign on the left to edit the code.">
42      /**
43       * Handles the HTTP GET method.
44       *
45       * @param request servlet request
46       * @param response servlet response
47       * @throws ServletException if a servlet-specific error occurs
48       * @throws IOException if an I/O error occurs
49       */
50      @Override
51      protected void doGet(HttpServletRequest request, HttpServletResponse
            response)
52              throws ServletException, IOException {
53          processRequest(request, response);
54      }
55
56      /**
57       * Handles the HTTP POST method.
58       *
59       * @param request servlet request
60       * @param response servlet response
61       * @throws ServletException if a servlet-specific error occurs
62       * @throws IOException if an I/O error occurs
63       */
64      @Override
65      protected void doPost(HttpServletRequest request, HttpServletResponse
            response)
66              throws ServletException, IOException {
67          processRequest(request, response);
68      }
69
70      /**
71       * Returns a short description of the servlet.
72       *
73       * @return a String containing servlet description
74       */
75      @Override
76      public String getServletInfo() {
77          return "Short description";
78      }// </editor-fold>
79
80  }
```

El *servlet* galetesServlet rebrà el nom d'usuari que el navegador web li envia mitjançant un formulari, crearà la galeta i emmagatzemarà el nom d'usuari dins d'aquesta. A més a més, crearà un altre formulari amb un botó.

> Una **galeta** (més coneguda per la seva denominació anglesa, ***cookie***) és un fragment d'informació enviat des d'un servidor de pàgines web a un navegador que pot ésser retornada pel navegador en posteriors accessos a aquest servidor.

**Tipus de galetes**

Existeixen molts tipus de galetes. Sempre es generen de la mateixa manera, però es diferencien en el seu ús. Per exemple, existeixen galetes de sessió, galetes persistents, galetes segures (https), galetes només http, i inclús galetes *zombie*. Aquestes últimes s'emmagatzemen en diversos llocs de l'aplicació, fins i tot en objectes *flash*, perquè si s'esborren es tornin a generar.

Però anem pas a pas. Primer de tot heu de recuperar l'usuari enviat pel navegador web. Aquest usuari s'emmagatzema a l'objecte `request`, i amb la funció `getParameter(nom_parametre)` podeu recuperar-lo:

```
1 String n = request.getParameter("userName");
```

Una vegada teniu la informació que voleu emmagatzemar a la galeta, heu de crear un objecte de tipus *cookie*, i els seus paràmetres de creació són els següents:

- *name*: correspon al nom de la galeta.
- *value*: correspon a la informació que es vol emmagatzemar.

```
1 Cookie ck = new Cookie("usuari", n);
```

Ja teniu creada la galeta, que pot ser de dos tipus:

- **Persistent** (*persistent cookie*): no s'esborra quan l'usuari tanca el navegador, sinó que la galeta té un temps màxim de vida i s'esborra quan arriba el moment.
- **No persistent** (*non-persistent cookie*): la galeta s'esborra quan l'usuari tanca el navegador.

Per defecte, la galeta es crea del tipus no persistent. Si voleu crear una galeta de l'altre tipus s'ha d'utilitzar el mètode de l'objecte *cookie* anomenat `setMaxAge(temps_en_millisegons)`.

Per exemple, per emmagatzemar una galeta al navegador durant 24 hores escriuríem:

```
1 ck.setMaxAge(24 * 60 * 60);  // 24 hores * 60 minuts * 60 segons
```

Si volguéssim esborrar una galeta en aquest mateix moment, sense haver d'esperar que l'usuari tanqui el navegador o que li arribi el temps establert, podeu posar-li un temps igual a zero, i la galeta s'esborra en zero mil·lisegons.

```
1 ck.setMaxAge(0);  //esborra la galeta
```

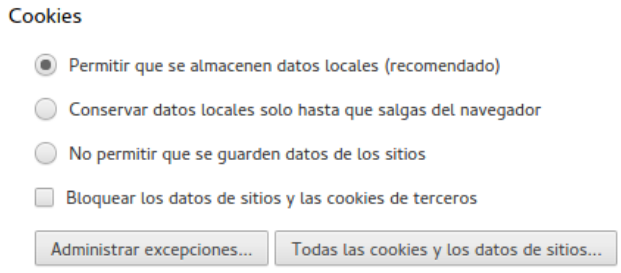
Una vegada s'ha creat la galeta, heu d'enviar-la al navegador perquè aquest sigui qui l'emmagatzemi dins del disc dur client.

```
response.addCookie(ck);
```

Com podeu veure, la utilització de galetes és una tècnica molt senzilla per guardar informació a l'ordinador. S'utilitza bàsicament per mantenir l'estat de l'aplicació en el costat client.

El desavantatge que us podeu trobar en fer servir les galetes és que el navegador té una opció per evitar que les pàgines web les emmagatzemin. Per exemple, al navegador web Chrome podeu anar a *Configuració / Mostrar configuració avançada / Privacitat / Configuració del contingut* per modificar el seu comportament quan un servidor vulgui enviar-li una galeta (vegeu la figura 3.2).

**Figura 3.2.** Configuració de les galetes del navegador Chrome




Com heu pogut comprovar, el funcionament de les galetes és senzill:

1. L'usuari demana la pàgina al servidor web.

2. El servidor web li envia la pàgina amb una galeta.

3. L'usuari envia la galeta cada vegada que es comunica amb el servidor web.

Arribats a aquest punt, la galeta està emmagatzemada al navegador client. Ara veurem com recuperar-la i com mostrar el seu contingut. Ho farem modificant el *servlet* GaletesServlet afegint un botó que enviï a l'usuari a un altre *servlet*, que serà qui recuperi el valor de la galeta creada.

El codi s'afegeix després que s'hagi creat la galeta i s'hagi enviat a l'usuari (vegeu la figura 3.3):

```
out.print("<form action='galetesServlet2'>");
out.print("<input type='submit' value='anar'>");
out.print("</form>");
```

**FIGURA 3.3.** Galeta emmagatzemada al navegador


Aquest botó enviarà l'usuari al *servlet* galetesServlet2, que recuperà la galeta i mostrarà el seu contingut en una pàgina web.

Cal crear el *servlet* GaletesServlet2 de la mateixa manera que heu creat anteriorment el *servlet* GaletesServlet. La implementació del seu codi és la següent:

```
1  @WebServlet(name = "galetesServlet2", urlPatterns = {"/galetesServlet2"})
2  public class GaletesServlet2 extends HttpServlet {
3
4      /**
5       * Processes requests for both HTTP GET and POST
6       * methods.
7       *
8       * @param request servlet request
9       * @param response servlet response
10      * @throws ServletException if a servlet-specific error occurs
11      * @throws IOException if an I/O error occurs
12      */
13     protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
14             throws ServletException, IOException {
15         response.setContentType("text/html;charset=UTF-8");
16         try (PrintWriter out = response.getWriter()) {
17             /* TODO output your page here. You may use following sample code.
                   */
18             out.println("<!DOCTYPE html>");
19             out.println("<html>");
20             out.println("<head>");
21             out.println("<title>Servlet galetes2</title>");
22             out.println("</head>");
23             out.println("<body>");
24             out.println("<h1>Accedim a la galeta per veure el nom d'usuari.</h1
                   >");
25             Cookie ck[]=request.getCookies();
26             out.print("Hola "+ck[0].getValue());
27             out.println("</body>");
28             out.println("</html>");
29         }
30     }
31
32     // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click
           on the + sign on the left to edit the code.">
33     /**
34      * Handles the HTTP GET method.
35      *
36      * @param request servlet request
37      * @param response servlet response
```

```
38      * @throws ServletException if a servlet-specific error occurs
39      * @throws IOException if an I/O error occurs
40      */
41     @Override
42     protected void doGet(HttpServletRequest request, HttpServletResponse
            response)
43             throws ServletException, IOException {
44         processRequest(request, response);
45     }
46 
47     /**
48      * Handles the HTTP POST method.
49      *
50      * @param request servlet request
51      * @param response servlet response
52      * @throws ServletException if a servlet-specific error occurs
53      * @throws IOException if an I/O error occurs
54      */
55     @Override
56     protected void doPost(HttpServletRequest request, HttpServletResponse
            response)
57             throws ServletException, IOException {
58         processRequest(request, response);
59     }
60 
61     /**
62      * Returns a short description of the servlet.
63      *
64      * @return a String containing servlet description
65      */
66     @Override
67     public String getServletInfo() {
68         return "Short description";
69     }// </editor-fold>
```

Una vegada que està creat, podeu recuperar la galeta de la següent manera:

```
1 Cookie ck[]=request.getCookies();
```

Tots els navegadors proporcionen una manera de veure les galetes que ha emmagatzemat una pàgina web. En concret, a Google Chrome hi podem accedir clicant el botó dret a la pàgina i seleccionant l'element de menú anomenat 'inspecciona'. S'obrirà una finestra on haurem de clicar la pestanya 'recursos' per poder veure tota la informació que ens proporciona la pàgina, incloent-hi les galetes.

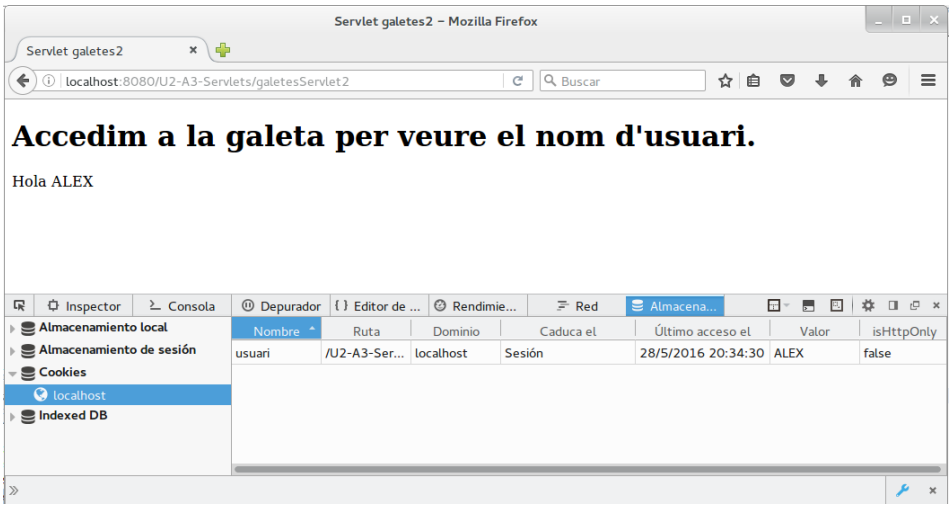
Adoneu-vos que el mètode `getCookies()` retorna totes les galetes que té el navegador de la nostra pàgina web. Així, si teniu més d'una galeta heu de fer un bucle per trobar la que us interessi. En el nostre cas no cal, només n'hem emmagatzemat una.

Així, podeu obtenir el valor de la galeta emmagatzemada fent ús del mètode `getValue()`.

```
1 out.print("Hola "+ck[0].getValue());
```

El resultat de l'execució del codi anterior el podeu veure a la figura 3.4.

FIGURA 3.4. Mostrar el contingut d'una galeta




### 3.1.1 Altres maneres d'enviar informació al client

Us podeu trobar que el navegador web no accepti galetes. En aquest apartat veureu unes tècniques no tan sofisticades com les galetes que us poden ajudar a emmagatzemar informació en el costat client.

#### Utilitzant un camp ocult

Aquesta tècnica empra un camp ocult d'un formulari per enviar informació a un altre *servlet*.

Utilitzant el mateix projecte Maven, crearem una pàgina web anomenada textOcult.html (*File / New File / HTML5 / HTML File*). Aquesta pàgina és idèntica a la pàgina galetes.html, únicament heu de canviar el contingut de la propietat `action` de l'etiqueta `formulari`:

```
1 <form action="TextOcultServlet" method="post">
```

Vegeu que el *servlet* que rep la petició de la pàgina web ha canviat. Llavors heu de crear aquest *servlet* seguint els mateixos passos que vam utilitzar en crear el *servlet* GaletesServlet.

Una vegada teniu el *servlet* creat, recolliu l'usuari que ens envien per paràmetre:

```
1 String n = request.getParameter("userName");
```

Seguidament, creem un formulari amb un camp de text ocult i emmagatzemem la informació dins d'aquest camp. L'usuari rebrà aquesta informació, però no la veurà.

```
1 out.print("<form action='TextOcultServlet2'>");
2 out.print("<input type='hidden' name='nom' value='" + n + "'>");
3 out.print("<input type='submit' value='anar'>");
4 out.print("</form>");
```

L'usuari només veurà un botó que li permet anar a una altra pàgina web del programa (vegeu la figura 3.5). Aquesta pàgina web serà un altre *servlet*, anomenat TextOcultServlet2, que rebrà el paràmetre ocult.

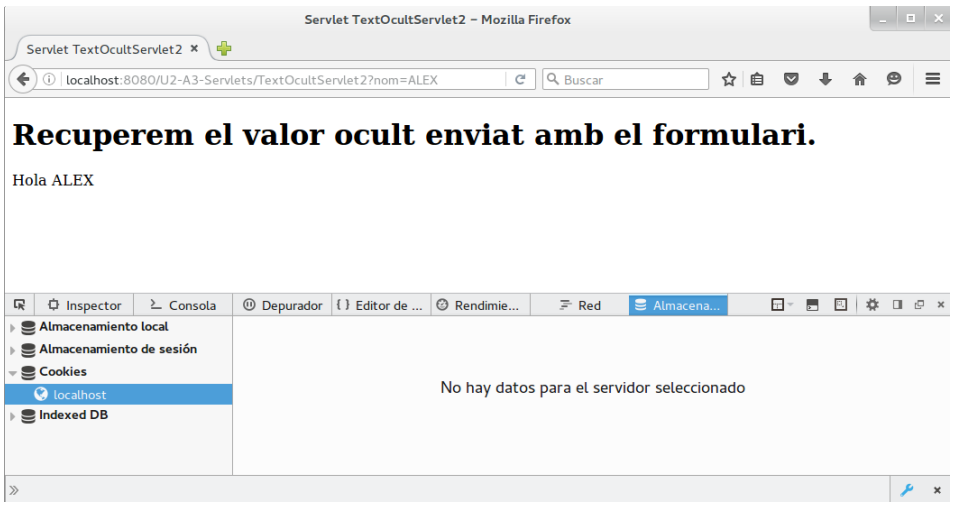
**Figura 3.5.** Pàgina web amb un text ocult


Finalment, només falta crear el *servlet* TextOcultServlet2 per rebre la informació que envia el navegador client sense que aquest ho sàpiga. Crearem el *servlet* de la mateixa manera que vam crear els *servlets* anteriors i modifiquem la funció `processRequest` perquè recuperi el paràmetre ocult del formulari i el mostri per pantalla.

```
1 out.println("<h1>Recuperem el valor ocult enviat amb el formulari.</h1>");
2 String n = request.getParameter("nom");
3 out.print("Hola " + n);
```

A la figura 3.6 podeu veure la pàgina que envia el *servlet* TextOcultServlet2.

**Figura 3.6.** Pàgina que mostra un text ocult



### Creant un URL amb paràmetres

Aquesta tècnica empra un URL amb paràmetres per enviar informació a un altre *servlet*.

Utilitzant el mateix projecte Maven, crearem una pàgina web anomenada reescripturaURL.html (*File / New File / HTML5 / HTML File*). Aquesta pàgina és idèntica a la pàgina galetes.html, únicament heu de canviar el contingut de la propietat `action` de l'etiqueta `formulari`:

```
<form action="ReescripturaURLServlet" method="post">
```

Vegeu que el *servlet* que rep la petició de la pàgina web ha canviat. Llavors heu de crear aquest *servlet* seguint els mateixos passos que en la creació del *servlet* GaletesServlet.

Una vegada teniu el *servlet* creat, recolliu l'usuari que ens envien per paràmetre:

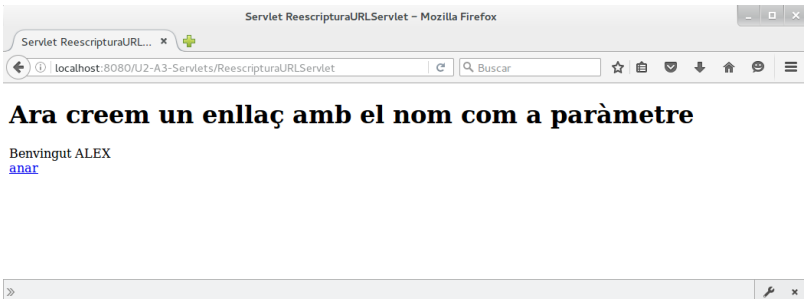
```
String n = request.getParameter("userName");
```

Seguidament, creem un URL i li afegim un paràmetre de tipus `GET`. La pàgina de l'usuari rebrà aquesta informació, i quan faci clic enviarà sense saber-ho el paràmetre a l'URL destí.

```
String n = request.getParameter("userName");
out.print("<br><a href='ReescripturaURLServlet2?nom="+n+"'>anar</a>");
```

L'usurari només veurà un enllaç que li permet anar a una altra pàgina web del programa (vegeu la figura 3.7). El destí d'aquest enllaç serà un altre *servlet*, anomenat ReescripturaURLServlet2, que rebrà el paràmetre.

**FIGURA 3.7.** Pàgina web amb un paràmetre a l'enllaç




Finalment, només ens falta crear el *servlet* ReescripturaURLServlet2 per rebre la informació que ens envia el navegador client mitjançant l'URL. Crearem el *servlet* de la mateixa manera que heu creat els *servlets* anteriors i modifiquem la funció `processRequest` perquè recuperi el paràmetre de l'URL i el mostri per pantalla.

```
out.println("<h1>Recuperem el valor enviat amb la URL.</h1>");
String n = request.getParameter("nom");
out.print("Hola " + n);
```

A la figura 3.8 podeu veure la pàgina que envia el *servlet* ReescripturaURLServlet2.

**FIGURA 3.8.** Pàgina que mostra el paràmetre enviat per l'URL


## 3.2 L'usuari a la sessió

En aquest apartat s'explicarà com guardar i recuperar informació de sessió. L'usuari emplenarà un formulari web. Les dades d'aquest formulari s'enviaran a un *servlet* que les emmagatzemarà a sessió. Un altre *servlet* consultarà la sessió i mostrarà les dades emmagatzemades.

Utilitzarem el mateix projecte Maven. Creeu una nova pàgina HTML, amb el programa NetBeans, anomenada sessio.html (*File / New File / HTML5 / HTML File*), on afegirem les dades d'un formulari web HTML. El codi de la pàgina pot ser semblant a aquest:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Sessió</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>Emmagatzemar l'usuari a la sessió.</h1>
        <h2>Introdueix el teu nom:</h2>
        <form action="SessioServlet" method="post">
            <label for="nom"> Nom:</label>
            <input id="nom" type="text" name="userName"/>  <br/>
            <input type="submit" value="enviar"/>
        </form>
    </body>
</html>
```

Com podeu veure, el formulari demana el seu nom a l'usuari de la pàgina web. Aquest nom serà enviat al *servlet* SessioServlet, que el recollirà i l'emmagatzemarà a sessió.

> Una **sessió** serveix per emmagatzemar informació entre diferents peticions HTTP.

En moltes ocasions ens trobarem amb el problema de compartir l'estat (dades d'usuari) entre un conjunt ampli de pàgines de la nostra aplicació. La classe `HttpSession`, que s'encarrega d'administrar la sessió, té una estructura de diccionari (*HashMap*) i permet emmagatzemar qualsevol tipus d'objecte de tal manera que pugui ser compartit per les diferents pàgines de l'aplicació.

El *servlet* SessioServlet és l'encarregat d'utilitzar la classe `HttpSession` per emmagatzemar el nom d'usuari dins d'ella. Així, crearem el *servlet*, tal com heu creat els anteriors, i modificarem la funció `processRequest` per tal d'afegir aquesta funcionalitat.

```
1 HttpSession session=request.getSession();
2 String n = request.getParameter("userName");
3 session.setAttribute( "nom",n);
4
5 out.print("<br><a href='SessioServlet2'>anar</a>");
```

Primer de tot, obteniu la sessió mitjançant la funció `getSession()` de l'objecte `Request`. Aquesta funció crea una nova sessió si no existeix una sessió creada prèviament. Si ja hi ha una sessió creada, s'utilitzarà la mateixa per poder manipular, si es vol, les dades que contingui.

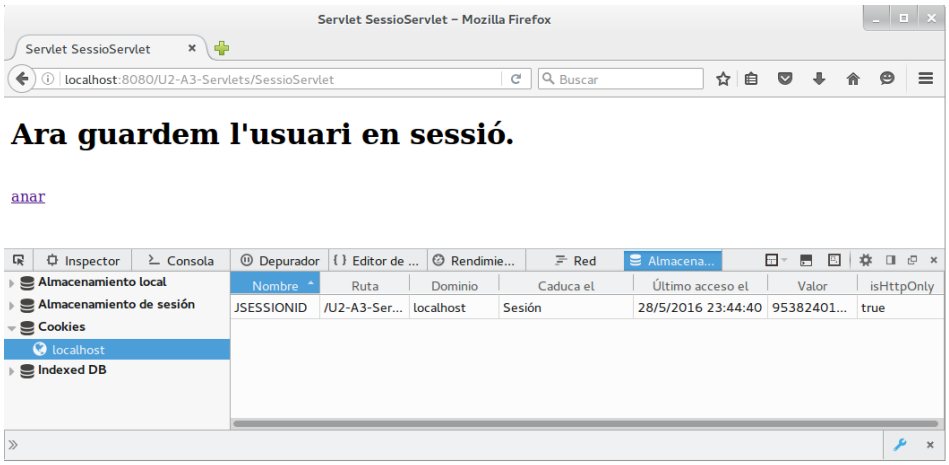
Una vegada teniu la sessió, afegiu el nom de l'usuari que s'ha recuperat de la petició. Per afegir el nom de l'usuari s'utilitza la funció `setAttribute`. Aquest mètode rep dos paràmetres: el primer correspon a l'identificador (nom) amb el qual podrem obtenir les dades emmagatzemades. El segon paràmetre correspon a l'objecte (valor) que es vol emmagatzemar.

Altres mètodes interessants de l'objecte sessió:

- `getCreationTime()`: retorna quan va ser creada la sessió, mesurada en mil·lisegons, des del dia 1 de gener de 1970.
- `invalidate()`: invalida la sessió i desvincula tots els objectes emmagatzemats. Recordeu que el procés *Garbage collection* recull tots els objectes que no estan vinculats a cap variable i els elimina.
- `getId()`: retorna l'identificador de la sessió.

Però on es guarda la sessió? A diferència de les galetes, la sessió es guarda en el costat del servidor. Una vegada s'ha creat la sessió s'envia al navegador de l'usuari una galeta que serveix per identificar-li i associar-li el *HashMap* que s'acaba de construir perquè pugui emmagatzemar-hi informació (vegeu la figura 3.9). A aquest *HashMap* s'hi pot accedir des de qualsevol altra pàgina, i ens permet compartir informació.

**FIGURA 3.9.** Galeta de sessió




Normalment s'utilitzen tècniques criptogràfiques per emmagatzemar les dades en el servidor en forma de sessió. Així, es garanteixen la confidencialitat, la integritat i la autenticitat de les dades emmagatzemades.

La sessió és individual de cada usuari que es connecta a la nostra aplicació, i la informació no és compartida entre ells. Així doncs, cada usuari disposarà del seu propi *HashMap* on emmagatzemar la informació que resulti útil entre pàgines.

> No s'ha d'abusar de l'**emmagatzematge d'objectes a sessió**, ja que si teniu molts usuaris concurrents estarem obligant el servidor a utilitzar molta memòria per emmagatzemar-los.

Ja heu vist com emmagatzemar els objectes a sessió, ara veurem com recuperar les dades. Crearem el *servlet* SessioServlet2 i afegirem un enllaç a la resposta del *servlet* anterior per tal que puguem accedir a aquest.

```
1 out.print("<br><a href='SessioServlet2'>anar</a>");
```

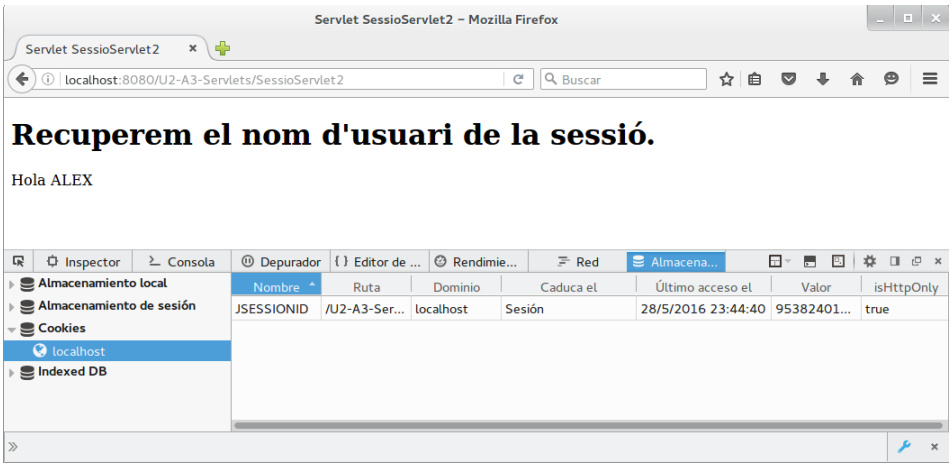
Modificarem la funció `processRequest` del *servlet* SessióServlet2 per accedir a la sessió i recuperar el nom d'usuari que s'ha emmagatzemat.

```
1 out.println("<h1>Recuperem el nom d'usuari de la sessió.</h1>");
2
3 HttpSession session = request.getSession();
4 String n = (String) session.getAttribute("nom");
5 out.print("Hola " + n);
```

Vegeu que per obtenir l'objecte de sessió utilitzem la mateixa funció que vam utilitzar en crear-la. Com que aquest client ja va crear una sessió el mètode `getSession` agafa la *cookie* amb la informació de la sessió i pot accedir al *HashMap* que es va crear amb els valors emmagatzemats.

Per obtenir un valor s'utilitza el mètode `getAttribute` i s'empra l'identificador del valor que es vol obtenir per recuperar-lo. Una vegada obteniu el valor ja el podrem usar, com en aquest cas, que el mostrem per pantalla. Vegeu la figura 3.10 per veure'n el resultat final:

**FIGURA 3.10.** Resultat d'obtenir una dada de sessió




## 3.3 Un formulari d'autenticació amb EJB

En aquest apartat s'explicaran l'autenticació i l'autorització utilitzant *servlets* i Enterprise Java Beans (EJB). Es recomana haver fet l'apartat anomenat "Formularis amb *servlets* i EJB", on s'explica el funcionament d'un EJB sense estat.

Es vol crear una aplicació que autentiqui l'usuari mitjançant un nom i una contrasenya. Una vegada s'ha autenticat, hi haurà una sèrie de mètodes d'un objecte EJB als quals tindrà permisos per accedir i alguns altres per als quals no tindrà permisos.

Creeu un nou *servlet* amb el programa NetBeans anomenat BibliotecaServlet.java (*File / New File / Web / Servlet*). No cal que afegiu la configuració del *servlet* al fitxer web.xml. En aquest cas, utilitzarem les anotacions per configurar-lo.

Crearem també un Enterprise Java Beans sense estat anomenat Biblioteca amb una interfície local associada anomenada BibliotecaLocal. Podeu crear les dues classes a la vegada utilitzant el programa Netbeans i accedint a *File / New File / Enterprise JavaBeans / SessionBean*.

La interfície BibliotecaLocal tindrà tres mètodes:

- `catalogar(String llibre)`
- `veureDisponibilitat(String llibre)`
- `demanarPrestec(String llibre)`

Aquests mètodes els sobreescriurà l'EJB sense estat Biblioteca. El mètode `catalogar` retornarà una cadena de text dient que s'ha catalogat el llibre, i el mètode `veureDisponibilitat` comparará si és un llibre Java. En el cas que sigui així, retornarà una cadena de text on s'informarà que està disponible, i en cas contrari s'informarà que no en queden còpies disponibles. Finalment, el mètode `demanarPrestec` retorna sempre *false*.

Un exemple de la seva codificació és la següent:

```
1 @Stateless
2 public class Biblioteca implements BibliotecaLocal {
```

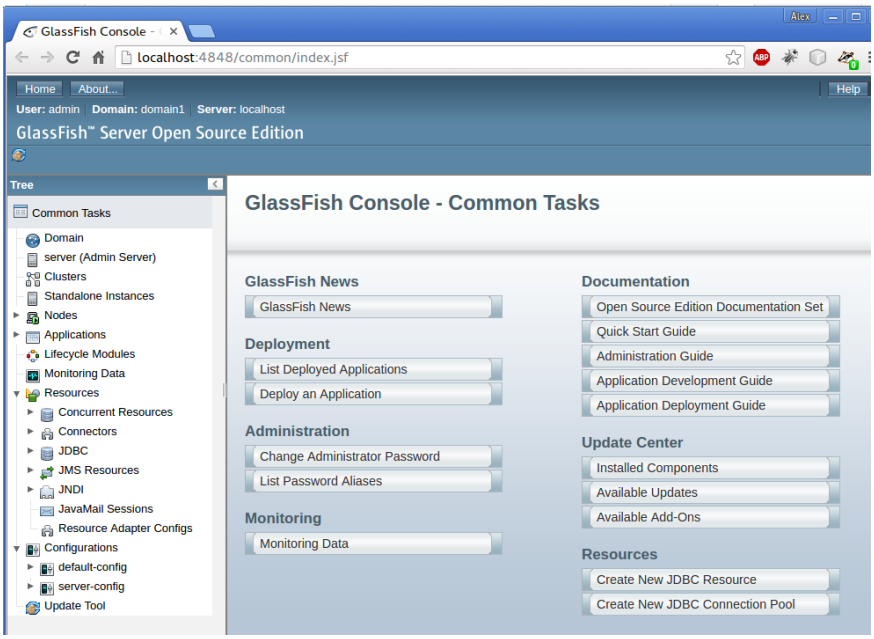
```
3
4     @Override
5     public String catalogar(String llibre){
6      return "Llibre " + llibre + " catalogat.";
7     }
8
9     @Override
10    public String veureDisponibilitat(String llibre){
11        if(llibre.equals("Java")){
12            return "Llibre " + llibre + " disponible.";
13        }
14        return "Llibre " + llibre + " no disponible.";
15    }
16
17
18    @Override
19    public Boolean demanarPrestec(String llibre){
20       return false;
21    }
22 }
```

Volem fer que el mètode `catalogar` només sigui accessible pel bibliotecari, que el mètode `veureDisponibilitat` estigui disponible per a tothom i que el mètode `demanarPrestec` no el pugui utilitzar ningú (ja que encara no s'ha implementat).

> L'**autenticació** és el procés mitjançant el qual el sistema s'assegura que un usuari és qui diu que és.

Per poder autoritzar l'accés als mètodes de l'EJB a un usuari heu de comprovar quin usuari està utilitzant l'aplicació. Podeu emprar els usuaris de GlassFish per portar a terme l'autenticació. Aquests usuaris els podeu configurar mitjançant la consola d'administració (web) del servidor (vegeu la figura figura 3.11).

**FIGURA 3.11.** Pàgina d'administració del servidor Glassfish

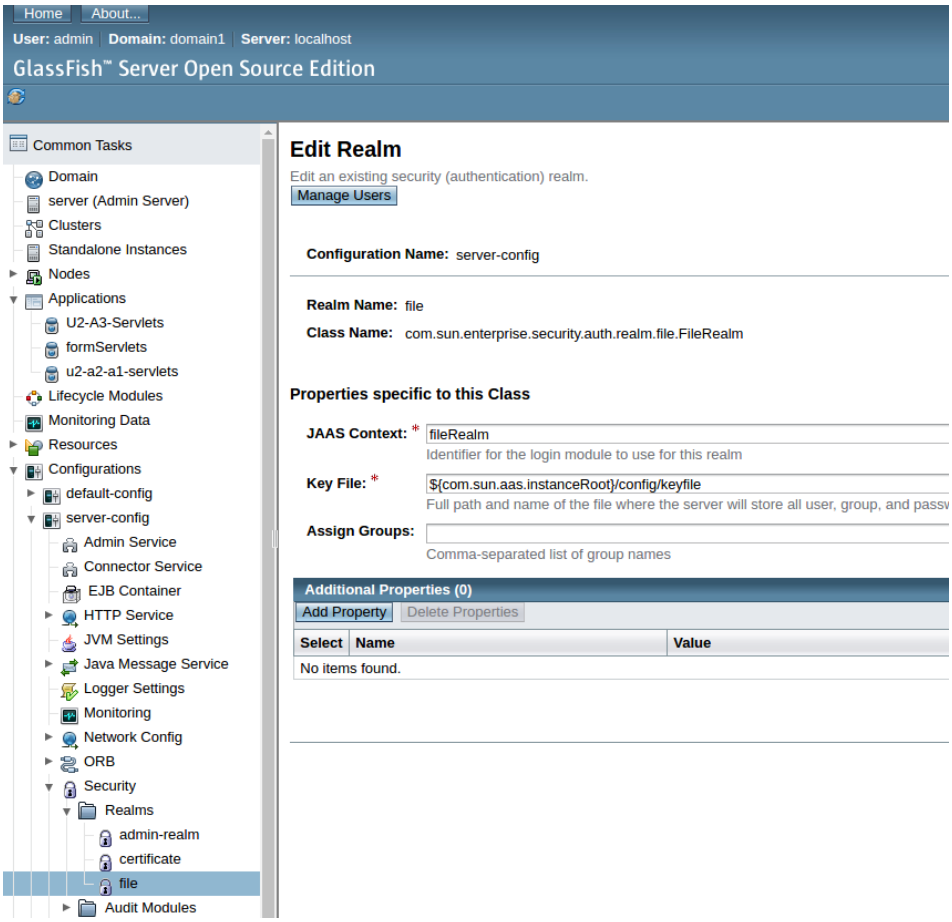



Per accedir a la consola podeu fer-ho posant a l'URL del navegador l'adreça localhost:4848/common/index.jsf, o bé amb el programa NetBeans accedint al

menú desplegat en fer clic amb el botó dret del ratolí damunt del servidor Glassfish i clicant a l'opció *View Domain Admin Console*.

Una vegada heu accedit a la consola, aneu al menú de l'esquerra i cliqueu a l'opció *Configurations / Servlet Config / Security / Realms / File* (vegeu la figura 3.12).

**FIGURA 3.12.** Menú de la consola d'administració per afegir-hi usuaris




Normalment els usuaris d'una aplicació es troben emmagatzemats en una base de dades, segurament, de la pròpia aplicació. No hem d'oblidar, però, que els servidors d'aplicacions ens proporcionen un mecanisme d'autenticació per a aplicacions petites.

Si us fixeu, el menú *Realm* té tres opcions:

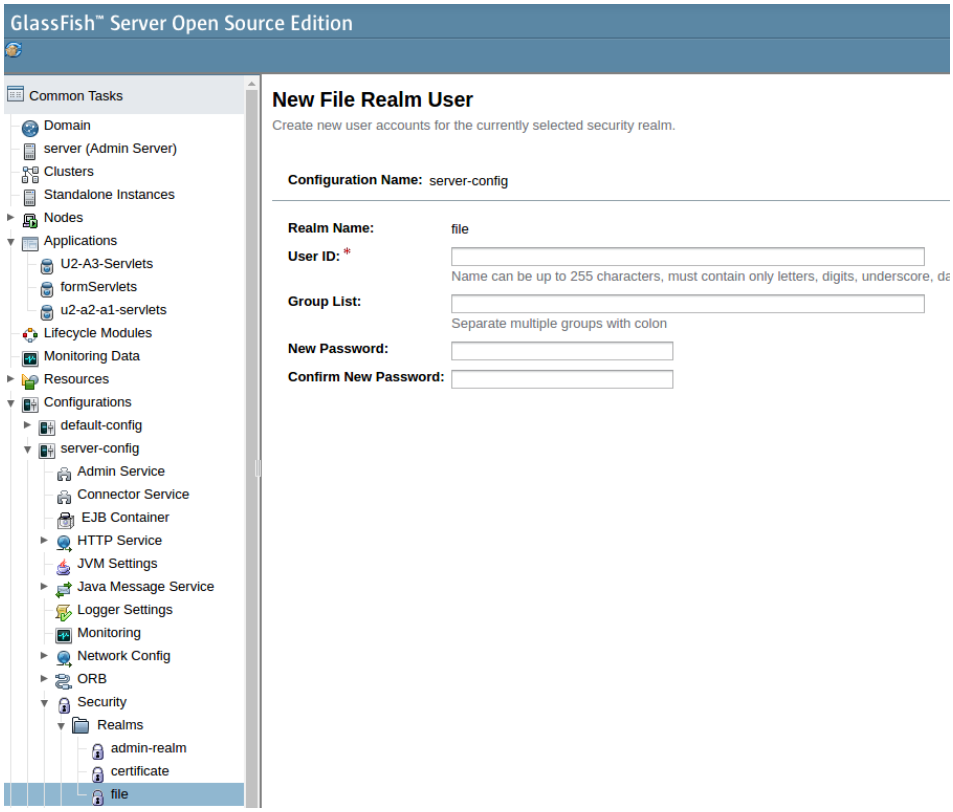
- ***admin-realm***: és un fitxer on s'emmagatzemen les credencials dels usuaris administradors. Aquest fitxer s'anomena admin-keyfile.
- ***certificate***: el servidor emmagatzema les credencials d'usuari en una base de dades de certificats. Quan s'utilitza aquesta opció, el servidor utilitza certificats amb HTTPS per autenticar els clients web. Per verificar la identitat d'un usuari en el domini certificat, el servei d'autenticació verifica un certificat X.509.
- ***file***: el servidor emmagatzema les credencials d'usuari local en un arxiu amb el nom de keyfile. El servei d'autenticació del servidor verifica la identitat de l'usuari mitjançant la comprovació d'aquest fitxer, que s'utilitza per a l'autenticació de tots els clients excepte per als clients de l'explorador web que utilitzen HTTPS i certificats.

> Un àmbit o domini (***realm***) és una política de seguretat definida per a un servidor web o aplicació. Conté una col·lecció d'usuaris, que poden o no poden ser assignats a un grup.

Vosaltres emmagatzemareu tots els usuaris dintre d'un fitxer al nostre servidor web, i llavors accedireu a la opció de menú *file*.

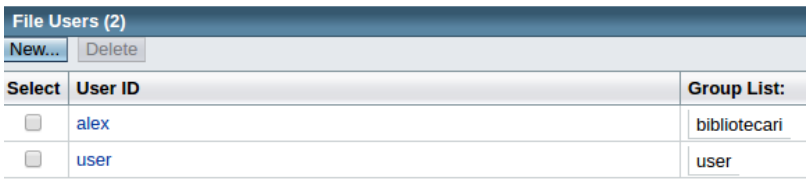
Tot seguit, procedireu a la creació d'usuaris clicant al botó *Manage users*. Una vegada heu accedit al llistat d'usuaris, el programa us permet crear de nous clicant al botó *new*. Si ho feu, apareixerà una pantalla com la que podeu veure a la figura 3.13.

**Figura 3.13.** Creació d'un usuari nou




Hi afegireu dos usuaris. El primer l'anomenareu *alex*, o amb qualsevol altre nom, i pertanyerà al grup *bibliotecari*. En canvi, el segon usuari pertanyerà al grup d'*user*, i el seu nom serà *user* (vegeu la figura 3.14).

**Figura 3.14.** Llistat d'usuaris del servidor Glassfish

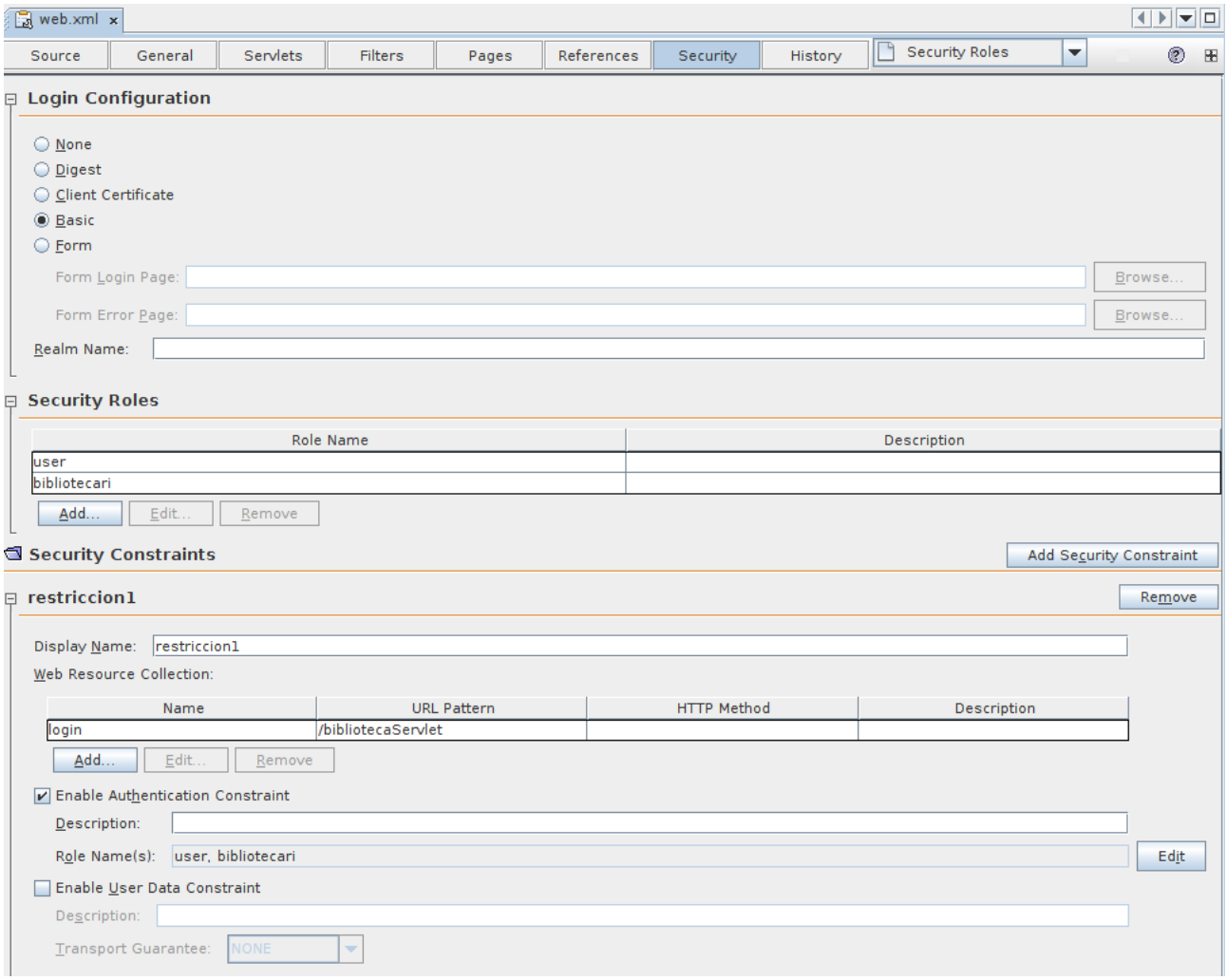



Una vegada definits els usuaris i els grups al servidor Glassfish heu de definir els rols dintre del fitxer de desplegament web.xml. Si no teniu aquest fitxer podeu

crear-lo amb el programa NetBeans. Aneu a *File / New File / Web / Standard Deployment Descriptor (web.xml)* i creeu-lo dins de la carpeta *WEB-INF*.

Aneu a l'apartat de seguretat del fitxer web.xml i afegiu-hi els rols, tal com podeu veure a la figura 3.15):

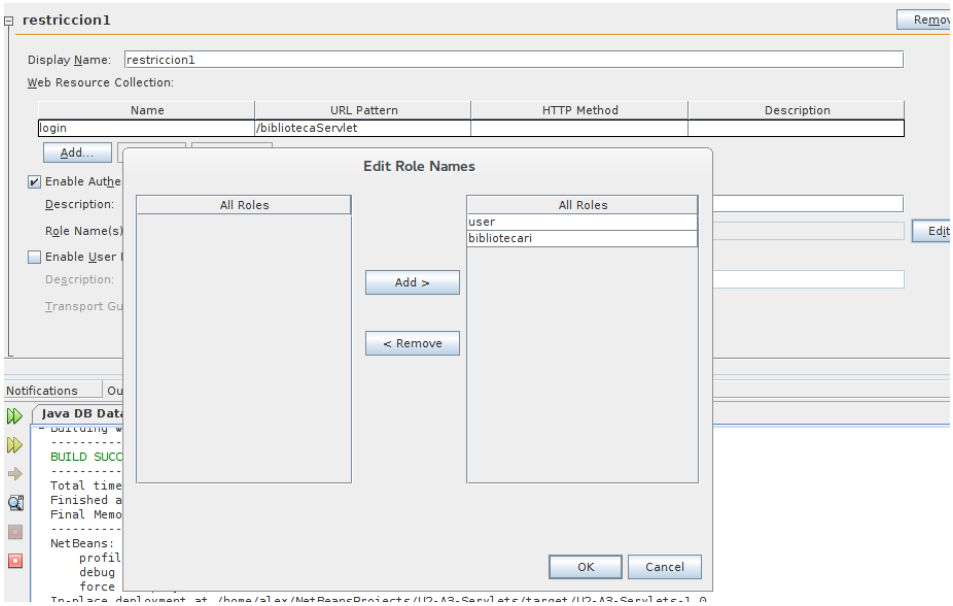
**FIGURA 3.15.** Configuració dels rols d'usuari al fitxer web.xml




A continuació cal especificar quin *servlet* de l'aplicació necessita autenticació. Per accedir a aquests *servlets* se'ls demanarà un usuari i una contrasenya. Per definir-los cal crear una restricció (*constraint*) nova en el fitxer web.xml.

A més a més, en la restricció s'ha d'especificar quins rols podran utilitzar aquest *servlet*. Cal seleccionar, mitjançant el botó *Edit* de l'opció *Enable autentication Constraint*, els rols creats anteriorment (vegeu la figura 3.16).

**FIGURA 3.16.** Configuració de l'autenticació




Finalment, haureu de tenir un fitxer web.xml com aquest:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="
    http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
    xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1
    .xsd">
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <security-constraint>
        <display-name>restriccion1</display-name>
        <web-resource-collection>
            <web-resource-name>login</web-resource-name>
            <description/>
            <url-pattern>/bibliotecaServlet</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <description/>
            <role-name>user</role-name>
            <role-name>bibliotecari</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
    </login-config>
    <security-role>
        <description/>
        <role-name>user</role-name>
    </security-role>
    <security-role>
        <description/>
        <role-name>bibliotecari</role-name>
    </security-role>
</web-app>
```

Per tal que funcioni, al servidor Glassfish s'ha d'activar el rol principal. Podeu anar a la web d'administració del servidor Glassfish i al menu *Server-config / Security* activar l'opció (vegeu la figura 3.17).

**FIGURA 3.17.** Activació del rol principal al servidor Glassfish


Arribats a aquest punt, ja heu acabat de configurar l'autenticació per al *servlet* BibliotecaServlet. A partir d'ara, cada vegada que accediu al *servlet* us demanarà que proporcioneu un usuari i una contrasenya vàlids.

Seguidament, definireu els rols dels usuaris que poden executar els diferents mètodes de l'EJB biblioteca. Aquest rols han de coincidir amb els rols definits a l'arxiu web.xml anterior.

L'**autorització** defineix quin usuari o grups d'usuari poden executar un recurs determinat. Tot i que tinguin permís d'accés a un recurs, potser no poden utilitzar tot el recurs, sinó només una part.

És essencial identificar el sistema o els usuaris que accedeixen a les aplicacions i proporcionar l'accés o la negació dels recursos dins de l'aplicació basada en alguns criteris. No tots els usuaris han de tenir els drets d'accés a dades sensibles i cal que hi hagi algun mecanisme d'identificació per restringir-ho.

En el vostre cas, el recurs EJB anomenat biblioteca poden utilitzar-lo, mitjançant el *servlet* BibliotecaServlet, els usuaris que pertanyin als rols *bibliotecari* o *users*. Però, en concret, voleu que el mètode `catalogar` només sigui accessible per al bibliotecari, que el mètode `veureDisponibilitat` estigui disponible per a tothom i que el mètode `demanarPrestec` no el pugui utilitzar ningú (ja que encara no s'ha implementat).

L'autorització s'ha de fer a nivell d'EJB. Heu d'afegir, mitjançant anotacions, quins mètodes poden executar els diferents rols del sistema. Les anotacions que podeu fer servir són les següents:

- `DeclareRoles`: indica que l'EJB acceptarà els rols definits amb aquesta anotació.
- `RolesAllowed`: indica quin mètode pot ser accessible per un rol determinat.
- `PermitAll`: indica que aquest mètode és accessible per a tothom.
- `DenyAll`: indica que aquest mètode no és accessible per cap rol.

A continuació, modificareu la classe EJB `Biblioteca` per afegir-hi la informació anterior:

```
1  @DeclareRoles({"bibliotecari"})
2  @Stateless
3  public class Biblioteca implements BibliotecaLocal {
4
5     @RolesAllowed({"bibliotecari"})
6     @Override
7     public String catalogar(String llibre){
8      return "Llibre " + llibre + " catalogat.";
9     }
10
11    @PermitAll
12    @Override
13    public String veureDisponibilitat(String llibre){
14        if(llibre.equals("Java")){
15            return "Llibre " + llibre + " disponible.";
```

```
        }
        return "Llibre " + llibre + " no disponible.";
    }

    @DenyAll
    @Override
    public Boolean demanarPrestec(String llibre){
       return false;
    }

}
```

D'aquesta manera, la funció `catalogar` només la podran executar els usuaris que pertanyin al rol *bibliotecari*, la funció `veureDisponibilitat` podrà executar-la tothom que tingui accés a l'EJB i la funció `demanarPrestec` no podrà executar-la ningú.

Les versions anteriors d'EJB 3.0 no podien utilitzar anotacions i empraven un fitxer XML addicional per configurar el comportament. El fitxer s'anomena ejb-jar.xml i s'ha de crear a la mateixa carpeta que el fitxer de desplegament web.xml. Un exemple de declaració de rols EJB (que han de coincidir amb els declarats al fitxer web.xml) pot ser:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server
      9.0 EJB 3.0//EN" "http://www.sun.com/software/appserver/dtds/sun-ejb-
     jar_3_0-0.dtd">
<sun-ejb-jar>
  <security-role-mapping>
    <role-name>user</role-name>
    <group-name>user-group</group-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>bibliotecari</role-name>
    <group-name>bibliotecari-group</group-name>
  </security-role-mapping>
  <enterprise-beans/>
</sun-ejb-jar>
```

Ja heu acabat de donar l'autorització demanada als rols dels usuaris perquè puguin executar els mètodes apropiats de l'objecte EJB. A continuació implementareu el *servlet* BibliotecaServlet perquè executi uns mètodes o uns altres segons el rol de l'usuari.

Primer injectareu l'objecte EJB dintre del *servlet* per poder-lo utilitzar.

```
@EJB
private BibliotecaLocal b;
```

A continuació, modificareu la funció `processRequest` perquè mostri per pantalla el nom de l'usuari que ha accedit al *servlet* i que comprovi si l'usuari és *bibliotecari*.

```
out.println("<p>Usuari: " + request.getUserPrincipal() + "</p>");
if(request.isUserInRole("bibliotecari")){
    out.println("<p>" + b.catalogar("java") + "</p>");
    out.println("<p>" +  b.veureDisponibilitat("php") + "</p>");
}
```

Amb la funció `getUserPrincipal()` teniu accés al nom de l'usuari que s'ha registrat a l'aplicació, i la funció `isUserInRole(nomRol)` comprova si l'usuari té el rol enviat com a paràmetre de la funció. En el cas que l'usuari sigui bibliotecari s'executaran les funcions del component EJB `catalogar` i `veureDisponibilitat`.

En canvi, si l'usuari que s'ha registrat té el rol *user*, llavors no hauria d'accedir a `catalogar`, ja que no hi està autoritzat, però sí que podria accedir a `veureDisponibilitat`. El codi seria el següent:

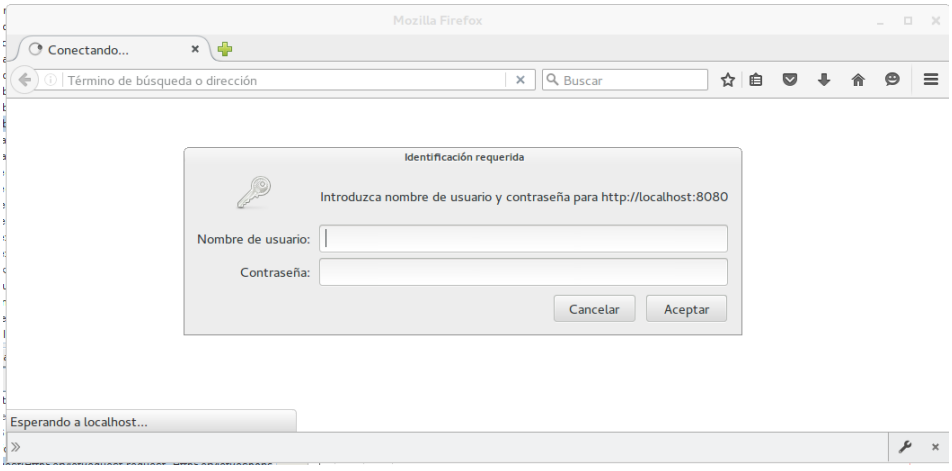
```
1 if(request.isUserInRole("user")){
2     out.println("<p>" + b.veureDisponibilitat("java") + "</p>");
3 }
```

Finalment, es mostra el que passa si qualsevol dels dos usuaris intenta accedir a la funció `demanarPrestec`. Aquesta funció està configurada de tal manera que cap dels dos usuaris hauria de poder-la executar. Si ho fan, el component EJB llença una excepció que s'ha de recollir i tractar.

```
1 if(b.demanarPrestec("java")){
2     out.println("<p> Mai es veurà aquest text.</p>");
3 }
```

En executar el *servlet* BibliotecaServlet us demana un usuari i una contrasenya (vegeu la figura 3.18).

**FIGURA 3.18.** Finestra d'autenticació per accedir al 'servlet' BibliotecaServlet




En utilitzar l'usuari amb rol *bibliotecari*, la informació que mostra es pot veure a la figura 3.19.

**FIGURA 3.19.** Execució dels mètodes autoritzats al rol 'bibliotecari'


En canvi, si s'utilitza l'usuari amb el rol *user*, la informació que mostra es pot veure a la figura 3.20.

**FIGURA 3.20.** Execució dels mètodes autoritzats al rol 'usuari'


Si us n'heu adonat, tant en l'execució del programa amb el rol *bibliotecari* com en l'execució del programa amb el rol *usuari*, el *servlet* ha mostrat per pantalla que no teniu permisos per accedir a un recurs. Això és degut al fet que intenteu accedir, tant en un cas com en l'altre, a la funció `demanarPrestec()`, i aquesta funció no es pot utilitzar. De fet, el servidor llança la següent excepció (podeu veure el *log* d'execució).

```
1 Información:   JACC Policy Provider: Failed Permission Check, context(U2-A3-
      Servlets/U2-A3-Servlets_internal)- permission(("javax.security.jacc.
      EJBMethodPermission" "Biblioteca" "demanarPrestec,Local,java.lang.String")
      )
2 Advertencia:   A system exception occurred during an invocation on EJB
      Biblioteca, method: public java.lang.Boolean cat.ioc.m7.u2.a3.servlets.
      Biblioteca.demanarPrestec(java.lang.String)
3 Advertencia:   javax.ejb.AccessLocalException: Client not authorized for this
      invocation
```

## 3.4 Què s'ha après?

Al acabar aquest apartat s'ha après a utilitzar correctament les diferents tècniques per guardar les preferències de l'usuari de l'aplicació. Les tècniques estudiades són les següents:

- *Cookies*
- Formularis amb informació oculta
- Reescriptura d'adreces web (URL)
- Sessions

A més a més, s'ha utilitzat els components de seguretat EJB per realitzar l'autenticació i l'autorització dels usuaris. Així, tot i que els usuaris poden accedir a l'aplicació mitjançant unes credencials pot ser, no tenen els permisos suficients per accedir a totes les parts del sistema.

En acabar aquest apartat, els estudiants ja estan preparats per començar les activitats proposades en aquest apartat i poden continuar amb el seu aprenentatge del llenguatge Java Web.