



Desenvolupament web en entorn servidor

CFGS.DAW.M07/0.16

Desenvolupament d'aplicacions web

Aquesta col·lecció ha estat dissenyada i coordinada des de l'Institut Obert de Catalunya.

Coordinació de continguts
Miguel Ángel Lozano Márquez

Redacció de continguts
Raúl Velaz Mayo
Àlex Salinas Tejedor
Mercedes Castellón Fuentes
Miguel Angel Lozano Márquez
Sergi Pérez Pérez

Primera edició: setembre 2016
© Departament d'Ensenyament

Dipòsit legal: DL B 29918-2017



Llicenciat Creative Commons BY-NC-SA. (Reconeixement-No comercial-Compartir amb la mateixa llicència 3.0 Espanya).

Podeu veure el text legal complet a

<http://creativecommons.org/licenses/by-nc-sa/3.0/es/legalcode.ca>

Introducció

Internet ha proporcionat la interconnexió de màquines que han fet possible la connexió i intercanvi de dades entre persones, entre organitzacions i entre dispositius.

Aquesta possibilitat, l'accés als grans amples de banda i amb diversos tipus de dispositius, ha provocat l'aparició d'un nou paradigma d'ús d'Internet que origina un creixement exponencial en les dades, tant si són estructurades com no.

Partíem de servidors com Apache o IIS (Internet Information Server de Microsoft) que liuren pàgines estàtiques, com per exemple, la pàgina d'informació general d'una empresa, programada amb HTML, CSS i pot ser, una part amb Java Script.

Avui en dia, tenint sentit pàgines com l'anterior, la necessitat de dades, ja sigui com a consumidors o com a productors, fa que les pàgines que sol·licitem des de el navegador hagin de ser dinàmiques, és a dir, construïdes amb les dades que donen resposta a la petició.

Aquestes pàgines no poden gestionar-les els servidors web, s'han de construir als servidors d'aplicacions, com Apache Tomcat, Oracle Glassfish, el servidor de PHP i IIS amb l'ampliació a tal efecte. Els llenguatges que es fan servir són Java Enterprise Edition (JEE) en els dos primers cassos, PHP en el tercer i .NET en el cas del de Microsoft.

En aquest mòdul farem servir Java Enterprise Edition (JEE) i el framework Spring MVC també amb Java. JEE i Spring MVC permeten desenvolupar de manera natural aplicacions en capes, fent independent la tecnologia del client i també on i com estan guardades les dades.

Aquesta modularitat ens permet construir aplicacions més fàcils de mantenir i més escalables, però també permet fer servir llibreries que ens proporcionen nivells alts de seguretat. Tot això, fa que la majoria d'empreses amb necessitat d'aquests tipus d'aplicacions facin servir JEE i Spring com a llenguatges de programació en entorn servidor.

A la unitat Introducció als llenguatges de servidor veureu com posar a punt tot l'entorn, amb NetBeans i el servidor d'aplicacions Oracle Glassfish; i com treballar amb llenguatges de servidor però encastrats dins del propi HTML. En aquest cas, donarem una ullada a PHP, i a JSP (Java Server Pages)

A la unitat Desenvolupament web en entorn servidor treballareu amb tecnologies de Java en el servidor que permeten recollir peticions des de el servidor, en aquest cas Servlets i Enterprise Java Beans (EJB)

A la unitat Generació dinàmica de pagines web podreu desenvolupar una aplicació web amb el framework Spring MVC que és l'òptim per seguir el patró MVC (Model View Controller) i l'estrucció en capes que separen perfectament la part vista (com presentem les dades i com interacciona l'usuari), de la part de negoci (quines regles, càlculs i restriccions s'apliquen) i de la part de dades (on i

com s'emmagatzemen les dades)

A la unitat Tècniques d'accés a dades veureu com podem relacionar-nos amb les dades des de les nostres aplicacions, fent servir Java Enterprise Edition directament o els frameworks Spring i Hibernate.

Moltes vegades, ens interessarà que les nostres aplicacions acceptin peticions diverses, no només des de navegadors, també des d'altres aplicacions, com per exemple, una companyia aèria que proporciona les dades a diversos cercadors de viatges.

En aquests casos, no podem saber la tecnologia emprada pel client i ens interessa publicar un servei web que doni resposta a aquests tipus de petició. A les unitats Serveis web amb Java EE 7 i Serveis web amb Spring, practicareu com escriure serveis web i com consumir-los amb JEE i amb Spring.

Resultats d'aprenentatge

En finalitzar aquest mòdul l'alumne/a:

Desenvolupament web en entorn servidor

1. Selecciona les arquitectures i tecnologies de programació web en entorn servidor, analitzant les seves capacitats i característiques pròpies.
2. Escriu sentències executables per un servidor web reconeixent i aplicant procediments d'integració del codi en llenguatges de marques.
3. Escriu blocs de sentències embedguts en llenguatges de marques, seleccionant i utilitzant les estructures de programació.
4. Desenvolupa aplicacions web embedgudes en llenguatges de marques analitzant i incorporant funcionalitats segons especificacions.

Generació dinàmica de pàgines web

1. Desenvolupa aplicacions web identificant i aplicant mecanismes per separar el codi de presentació de la lògica de negoci.

Tècniques d'accés a dades

1. Desenvolupa aplicacions d'accés a magatzems de dades, aplicant mesures per mantenir la seguretat i la integritat de la informació.

Serveis web. Pàgines dinàmiques interactives. Webs Híbrids

1. Desenvolupa serveis web analitzant el seu funcionament i implantant l'estructura dels seus components.
2. Genera pàgines web dinàmiques analitzant i utilitzant tecnologies del servidor web que afegeixin codi al llenguatge de marques.
3. Desenvolupa aplicacions web híbrids seleccionant i utilitzant llibreries de codi i dipòsits heterogenis d'informació.

Continguts

Desenvolupament web en entorn servidor

Unitat 1

Introducció als llenguatges de servidor

1. Entorns a punt per desenvolupar aplicacions web.
2. PHP i VDL a Java EE: Dades, estructures de control i arrays.

Unitat 2

Desenvolupament web en entorn servidor

1. Servlets.
2. Formularis amb Servlets i EJB.
3. Manteniment d'estat, autenticació i autorització amb Servlets i EJB

Generació dinàmica de pàgines web

Unitat 3

Generació dinàmica de pàgines web

1. Introducció a Spring i Spring MVC
2. Spring MVC. Aplicació Web
3. Spring MVC. Aprofundint en els controladors
4. Spring MVC. Altres aplicacions

Tècniques d'accés a dades

Unitat 4

Tècniques d'accés a dades

1. Accés a dades amb JDBC
2. Accés a dades amb Java Enterprise Edition
3. Accés a dades amb Spring i Hibernate

Serveis web. Pàgines dinàmiques interactives. Webs Híbrids

Unitat 5

Serveis web amb Java EE 7

1. Serveis web SOAP amb Java EE 7
2. Serveis web RESTful amb Java EE7. Escrivint serveis web
3. Serveis web RESTful amb Java EE7. Consumint serveis web

Unitat 6

Serveis web amb Spring

1. Serveis web SOAP amb Spring Web Services
2. Serveis web RESTful sobre Spring. Escrivint serveis web
3. Serveis web RESTful sobre Spring. Consumint serveis web

Introducció als llenguatges de servidor

Àlex Salinas Tejedor

Desenvolupament web en entorn servidor

Índex

Introducció	5
Resultats d'aprenentatge	7
1 Entorns a punt per desenvolupar aplicacions web	9
1.1 El vostre primer projecte web: 'Hola, Món'	10
1.1.1 Creació d'un projecte amb Maven	10
1.1.2 Exemple 'Hola, Món'	13
1.1.3 Pàgines dinàmiques i servidors d'aplicacions	21
1.1.4 Què s'ha après?	24
1.2 El vostre primer projecte PHP: 'Hola, Món'	25
1.2.1 Posar a punt el servidor web	25
1.2.2 Creació d'un projecte PHP en un servidor remot	29
1.2.3 Exemple 'Hola, Món'	33
1.2.4 Què s'ha après?	36
2 PHP i VDL a Java EE: dades, estructures de control i 'arrays'	37
2.1 Descrivint una persona amb PHP	38
2.1.1 Conversions de tipus	40
2.1.2 Àmbit de les variables	43
2.2 Gestionant un hotel amb PHP	45
2.2.1 Gestió de les taules del restaurant del hotel	46
2.2.2 Gestió de les habitacions de l'hotel	50
2.3 Descrivint una persona a JSP, JSTL	57
2.3.1 Sintaxi estàndard JSP	57
2.3.2 Sintaxi JSP Standard Tag Library (JSTL)	63
2.3.3 Calcular l'edat d'una persona	67
2.4 Gestionant un hotel amb JSP	69
2.4.1 Gestió de les taules d'un restaurant	69
2.4.2 Gestió de les habitacions d'un hotel	75
2.5 Què s'ha après?	81

Introducció

Al començament, l'èxit espectacular de la web va tenir lloc gràcies al protocol HTTP i al llenguatge HTML. El protocol HTTP és una implementació fàcil i senzilla d'un sistema de comunicacions que permet enviar qualsevol tipus de fitxers per la xarxa. Aquest protocol permet, a més, atendre milers de peticions reduint la potència dels servidors i, com a conseqüència, reduint els costos de desplegament dels serveis.

La web era un conjunt de pàgines estàtiques, documents simples d'informació que podien consultar-se o descarregar-se. A poc a poc va anar evolucionant fins a permetre desenvolupar pàgines dinàmiques que podien ser creades en funció de la petició enviada. Aquest mètode es va conèixer com a CGI (Common Gateway Interface). Els CGI defineixen una comunicació entre el client i el servidor mitjançant HTTP on els clients podien demanar informació als programes executats en el servidor. CGI especifica un estàndard per transferir dades entre el client i el programa. És un mecanisme de comunicació entre el servidor web i una aplicació externa. El resultat final de l'execució són objectes MIME. Aquestes aplicacions, que s'executen en el servidor, reben el nom de CGI.

Però els CGI tenen un punt feble: cada vegada que reben una petició, el servidor web comença un procés on executa el programa CGI. Com, d'altra banda, la majoria de CGI estaven escrits en algun llenguatge interpretat (Perl, Python, etc.), això implica una gran càrrega per a la màquina del servidor. A més, si la web té molts accessos al CGI, això suposa problemes greus de rendiment.

Per això es comencen a desenvolupar alternatives als CGI. Les solucions consisteixen a dotar el servidor d'un intèrpret d'algun llenguatge de programació (PHP, JSP, etc.) que permeti incloure el codi en les pàgines de manera que el servidor sigui qui ho executi, reduint així el temps de resposta.

A partir d'aquest moment es viu una explosió del nombre d'arquitectures i llenguatges de programació que ens permeten desenvolupar aplicacions web. Sorgeixen els llenguatges de programació integrats en el servidor que permeten interpretar instruccions incrustades a les pàgines HTML i un sistema d'execució de programes més enllaçat amb el servidor que no presenta els problemes de rendiment dels CGI.

En aquesta unitat abordarem amb detall uns llenguatges que permeten incrustar codi interpretable a les pàgines HTML i que el servidor tradueix a programes executables, JSP (Java Server Pages) i PHP (Hypertext Preprocessor). Però primer començarem la unitat explicant l'entorn de desenvolupament NetBeans, eina que ens ajudarà molt a l'hora d'escriure programes. S'explicarà com crear projectes fàcilment exportables utilitzant Maven i s'explicarà la diferència entre una pàgina web estàtica i una pàgina dinàmica.

En l'apartat “Entorns a punt per desenvolupar aplicacions web” estudiarem també el funcionament d'un servidor web i d'un servidor d'aplicacions, i veurem una pinzellada del protocol HTTP. S'acabarà l'apartat instal·lant i configurant un servidor PHP on puguem executar els exercicis realitzats.

En l'apartat “PHP i VDL a Java EE: dades, estructures de control i *arrays*” ens endinsarem en els llenguatges JSP i PHP. S'explicaran les peculiaritats d'ambdós llenguatges i s'explicaran les estructures de control i les funcions més utilitzades de les seves llibreries.

En acabar la unitat serem capaços d'instal·lar i configurar un servidor d'aplicacions, tant en JSP com en PHP, i crear aplicacions web senzilles que ens permetin gestionar petits negocis. 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 web.
- 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.
- Reconeix la sintaxi del llenguatge de programació que s'ha d'utilitzar.
- Escriu 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.

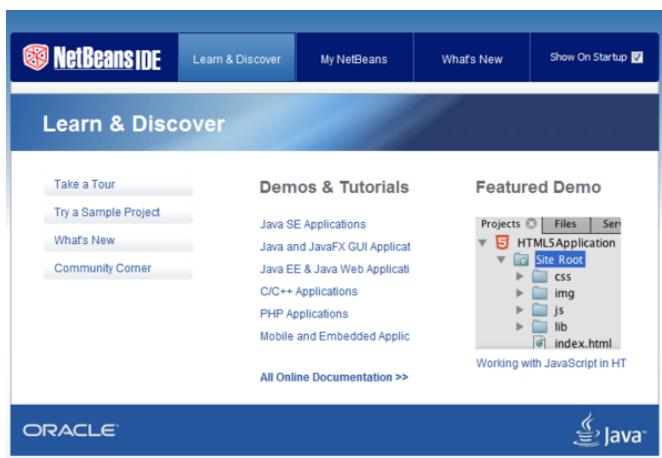
1. Entorns a punt per desenvolupar aplicacions web

Java és important. Els desenvolupadors web necessiten un coneixement exhaustiu del llenguatge que fa que les aplicacions web cobrin vida. Per poder començar a codificar en aquest llenguatge és important disposar d'un entorn de desenvolupament on pugueu realitzar totes les tasques associades a la programació:

- editar codi
- compilar-lo
- executar-lo
- depurar-lo

L'IDE (*Integrated Development Environment*) Netbeans us permet, d'una manera senzilla, elaborar programes complexos portant a terme totes les tasques abans esmentades. Per això, us animem a fer la primera activitat que tracta la instal·lació del IDE NetBeans (figura 1.1).

FIGURA 1.1. IDE Netbeans



En aquesta unitat posareu les bases de l'aprenentatge d'aplicacions web amb Java, tot abordant els següents continguts:

- Com crear un projecte web amb NetBeans i Maven.
- Quina diferència hi ha entre un servidor web i un servidor d'aplicacions.
- Com interacciona un navegador amb el servidor web.
- El protocol HTTP.
- Els diferents tipus de tecnologies que poden aparèixer en una aplicació web.

A més a més, es donarà una pinzellada al llenguatge de programació PHP, tot estudiant els següents aspectes:

- Com instal·lar i configurar un servidor PHP.
- Com configurar NetBeans per treballar amb un servidor PHP remot.
- Com interacciona un navegador web amb el servidor PHP.
- Com crear una pàgina senzilla amb aquest llenguatge.

Tots els conceptes tractats s'explicaran sempre partint de l'exemple. Quan acabeu aquesta unitat estareu preparats per endinsar-vos en la programació amb Java Servlets.

1.1 El vostre primer projecte web: 'Hola, Món'

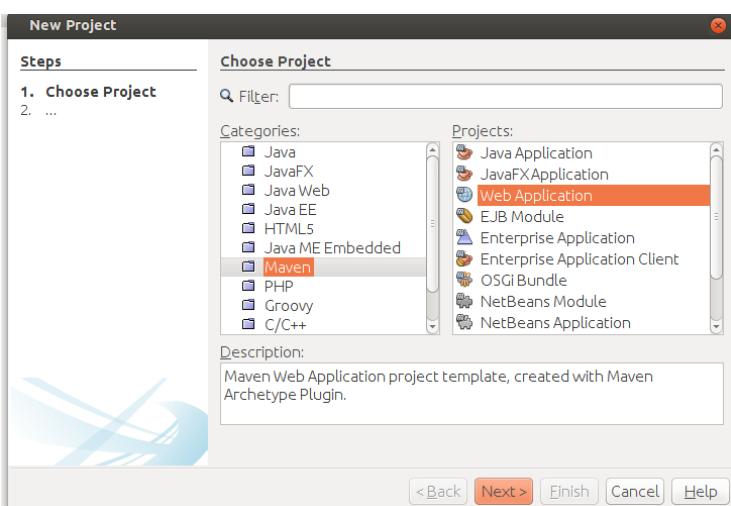
En aquest apartat veureu com crear un projecte web amb Netbeans utilitzant Maven, què és un servidor web, com interaccionen els navegadors amb els servidors web, quins tipus de servidors existeixen i quines tecnologies s'executen al servidor i quines al navegador.

1.1.1 Creació d'un projecte amb Maven

Utilitzareu Maven per administrar els projectes que creareu al llarg d'aquest mòdul. Maven és una eina d'administració de projectes que engloba tot el cicle de vida d'una aplicació, des de la seva creació fins als binaris amb el quals distribuir el projecte.

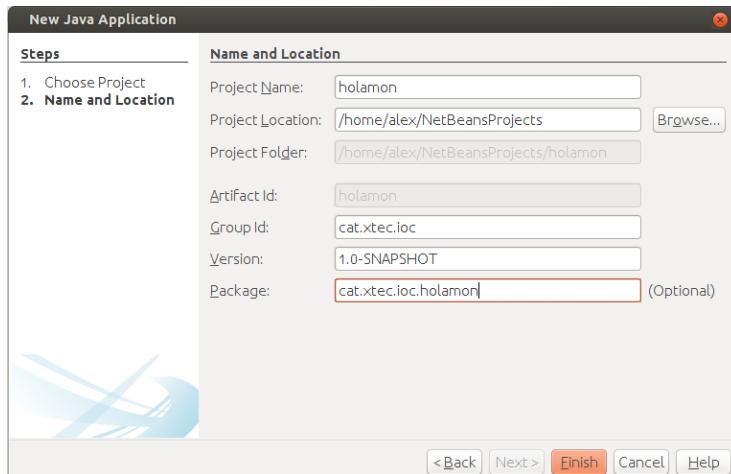
Per crear un nou projecte amb Maven i Netbeans és molt senzill, només heu d'anar a *File / New Project / Maven / Web Application* (vegeu la figura figura 1.2).

FIGURA 1.2. Creació d'una aplicació web utilitzant Maven



Podeu posar com a nom de projecte *Hola, Món*, i com a identificador de grup *cat.xtec.ioc*, tal com podeu veure a la figura figura 1.3

FIGURA 1.3. Propietats de Maven



Finalment, a l'última pantalla escollireu **GlassFish** com a servidor d'aplicacions.

Un projecte Maven es defineix mitjançant un fitxer anomenat **POM** (*Project Object Model*). Aquest fitxer s'utilitza per definir les instruccions per compilar el projecte, les dependències del projecte (llibreria), etc. En Maven, l'execució d'un fitxer POM sempre genera un **artefacte**. Aquest artefacte pot ser qualsevol cosa: un fitxer *jar*, un fitxer *swf* de Flash, un fitxer *zip* o el mateix fitxer *pom*.

Un exemple de fitxer POM podeu trobar-lo a la carpeta *ProjectFiles*:

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
2   /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM
3   /4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5
6   <groupId>cat.xtec.ioc</groupId>
7   <artifactId>holamon</artifactId>
8   <version>1.0</version>
9   <packaging>war</packaging>
10
11  <name>holamon</name>
12
13  <properties>
14    <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
15    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
16  </properties>
17
18  <dependencies>
19    <dependency>
20      <groupId>javax</groupId>
21      <artifactId>javaee-web-api</artifactId>
22      <version>7.0</version>
23      <scope>provided</scope>
24    </dependency>
25  </dependencies>
26
27  <build>
28    <plugins>
29      <plugin>
30        <groupId>org.apache.maven.plugins</groupId>

```

```

31             <configuration>
32                 <source>1.7</source>
33                 <target>1.7</target>
34                 <compilerArguments>
35                     <endorseddirs>${endorsed.dir}</endorseddirs>
36                 </compilerArguments>
37             </configuration>
38         </plugin>
39         <plugin>
40             <groupId>org.apache.maven.plugins</groupId>
41             <artifactId>maven-war-plugin</artifactId>
42             <version>2.3</version>
43             <configuration>
44                 <failOnMissingWebXml>false</failOnMissingWebXml>
45             </configuration>
46         </plugin>
47         <plugin>
48             <groupId>org.apache.maven.plugins</groupId>
49             <artifactId>maven-dependency-plugin</artifactId>
50             <version>2.6</version>
51             <executions>
52                 <execution>
53                     <phase>validate</phase>
54                     <goals>
55                         <goal>copy</goal>
56                     </goals>
57                     <configuration>
58                         <outputDirectory>${endorsed.dir}</outputDirectory>
59                         <silent>true</silent>
60                         <artifactItems>
61                             <artifactItem>
62                                 <groupId>javax</groupId>
63                                 <artifactId>javaee-endorsed-api</artifactId>
64                                 >
65                                 <version>7.0</version>
66                                 <type>jar</type>
67                             </artifactItem>
68                         </artifactItems>
69                     </configuration>
70                 </execution>
71             </executions>
72         </plugin>
73     </plugins>
74 </build>
</project>

```

Les etiquetes més importants del fitxer POM són:

- ***groupId***: és com el paquet del projecte. Normalment es posa el nom de l'empresa, ja que tots els projectes amb un *groupId* pertanyen a una sola empresa.
- ***artifactId***: és el nom del projecte.
- ***version***: nombre de versió del projecte.
- ***packaging***: tipus de fitxer generat en compilar el projecte Maven.

1.1.2 Exemple 'Hola, Món'

Una vegada s'ha creat el projecte, veieu que Maven genera un fitxer anomenat index.html dintre de la carpeta *Web Pages*, semblant a aquest:

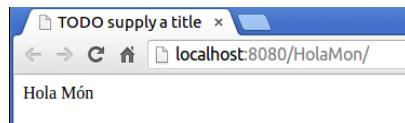
```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Start Page</title>
5     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6   </head>
7   <body>
8     <h1>Hello World!</h1>
9   </body>
10 </html>
```

Noteu que heu canviat el text “Hello World!” per “Hola, Món”. Aquest últim apareixerà com a contingut de la pàgina HTML.

Quan executeu el projecte anterior apareix en el navegador l'HTML anterior (vegeu la figura figura 1.4).

FIGURA 1.4. Execució del projecte 'Hola, Món'



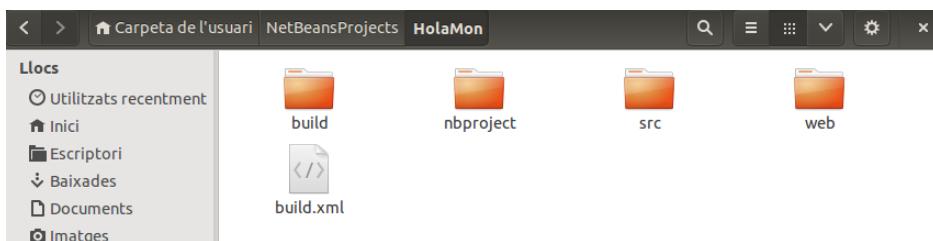
Fixeu-vos en la URL de la pàgina que ha executat el navegador: <http://localhost:8080/HolaMon/>.

Intenteu contestar a les següents preguntes:

- Què és <http://localhost:8080>?
- I *Hola, Món* què significa, a què fa referència?

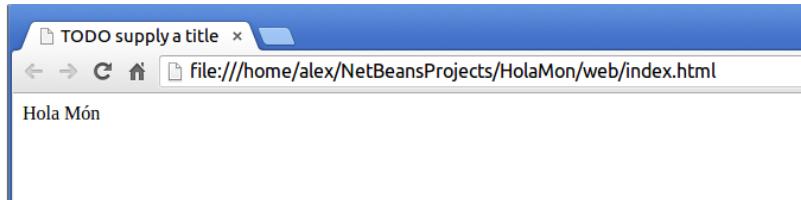
Abans de saber la resposta de les preguntes anteriors, cerqueu al vostre PC el fitxer index.html que heu creat anteriorment. Feu la cerca amb l'explorador de fitxers. Hauríeu de trobar-lo dintre de la carpeta *NetBeansProject/HolaMon/web* (vegeu la figura figura 1.5).

FIGURA 1.5. Fitxers del projecte 'Hola, Món'



Si obriu amb un navegador web el fitxer index.html es veurà correctament? Proveu-ho (vegeu la figura figura 1.6).

FIGURA 1.6. Execució del fitxer index.html directament al navegador



Observeu que el fitxer s'ha visualitzat exactament igual. No hi ha cap dubte que és la mateixa pàgina web. L'únic que ha canviat respecte a l'execució anterior és la URL del navegador. En aquest cas, es mostra la ruta des de l'arrel del sistema fins al fitxer que es veu en el navegador:

1 file:///home/alex/NetBeansProjects/HolaMon/web/index.html

Una pàgina **web estàtica** és aquella pàgina enfocada principalment a mostrar una informació permanent, on el navegent es limita a obtenir aquesta informació.

Llavors, quina és la diferència entre aquesta execució i l'execució de la figura figura 1.4?

Aquesta vegada el navegador ha llegit directament el fitxer, per això a la URL apareix la seva localització exacta. No hi ha cap altra intervenció. Ha obert el fitxer, ho ha interpretat (HTML) i l'ha tancat.

Per explicar que succeeix a la figura figura 1.4 ha arribat el moment de contestar a les preguntes abans formulades:

- Què és *http://localhost:8080?*
- I *Hola, Món* què significa, a què fa referència?

Com podeu suposar, el navegador no està obrint cap fitxer. El navegador, mitjançant el protocol HTTP, ha demanat un recurs a un servidor web. Aquest recurs és un fitxer HTML que ha llegit el servidor web i l'ha enviat al navegador a través de la xarxa fent servir el protocol HTTP.

Tecnologies al servidor

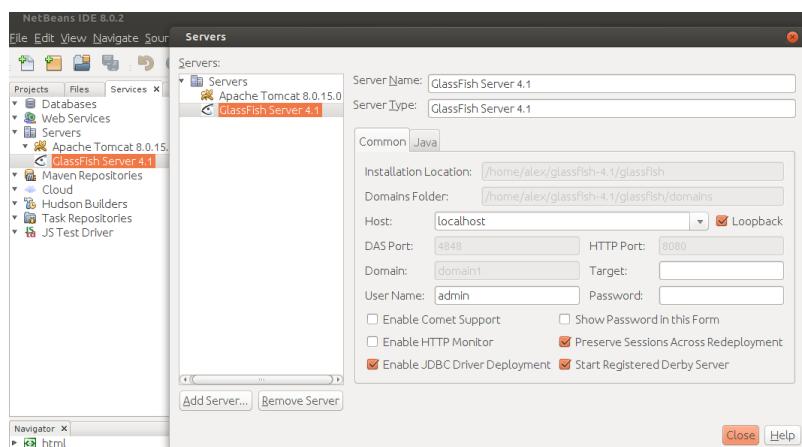
Es poden utilitzar diverses tecnologies per augmentar la potència del servidor més enllà de la seva capacitat de lliurar pàgines HTML; aquestes inclouen *scripts*, CGI, PHP, ASP, Java Servlets...

Un **servidor web** serveix contingut estàtic a un navegador, carrega un arxiu i el serveix a través de la xarxa al navegador d'un usuari. Aquest intercanvi d'informació entre el navegador i el servidor es produeix perquè parlen l'un amb l'altre mitjançant HTTP.

I el paràmetre :**8080?** Aquest paràmetre fa referència al port d'escolta del servidor web. Normalment, el paràmetre no el veieu quan accediu a una pàgina web

que està allotjada en un servidor públic, perquè s'accedeix a un port d'escola estàndard (el 80 per a pàgines no segures o el 443 per a pàgines segures). En aquest cas, l'entorn de desenvolupament NetBeans us proporciona un servidor integrat anomenat **GlassFish** que escolta les peticions pel port 8080. A més a més, com que està funcionant en el mateix PC des d'on s'està executant NetBeans, el nom del servidor és el mateix PC, o en altres paraules, *localhost*. Podeu veure les propietats si aneu a la finestra *Services* i després a les propietats del servidor (vegeu la figura figura 1.7).

FIGURA 1.7. Propietats del servidor GlassFish

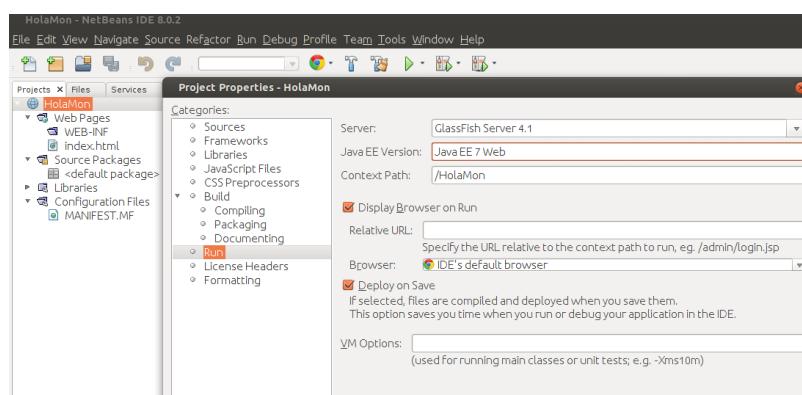


Si us hi fixeu, hi ha un paràmetre del servidor anomenat *HTTP Port* on s'especifica el port utilitzat per escoltar peticions HTTP, i un altre anomenat *HOST* on s'especifica que el servidor és el mateix PC (*localhost*). Recordeu que quan vàreu crear el projecte *Hola, Món* vàreu configurar-lo perquè utilitzés el servidor web GlassFish, que ve integrat amb NetBeans.

I *Hola, Món* què significa, a què fa referència?

Hola, Món identifica, dintre del servidor GlassFish, el **recurs web** que es vol visualitzar. Penseu que GlassFish pot publicar molts projectes alhora i que ha de tenir una manera per identificar cadascun. Una vegada ha estat identificat el projecte, GlassFish mostra la pàgina principal (index.html). Si veieu la figura figura 1.8 comprobareu on s'especifica el nom d'aquest recurs (*context path*).

FIGURA 1.8. Propietats del projecte 'Hola, Món'



Però perquè mostra la pàgina *index.html*? Per defecte, els servidors web cerquen aquest fitxer, però podeu canviar aquest comportament creant un arxiu **web.xml** dintre de la carpeta *WEB-INF*. Per crear-lo podeu anar a *New / Other / Standard Deployment Descriptor*, i aquí li podeu dir un altre arxiu d'inici de l'aplicació amb, per exemple:

```

1 <welcome-file-list>
2   <welcome-file>página_principal.html</welcome-file>
3 </welcome-file-list>
```

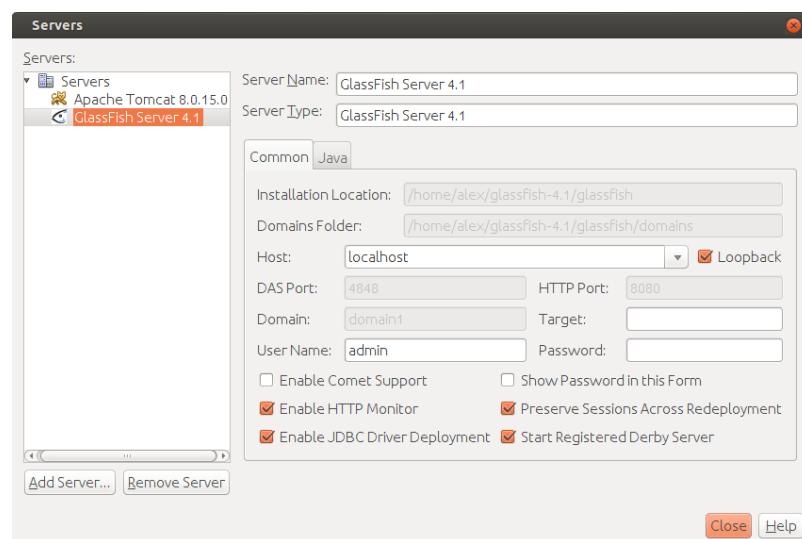
HTTP disposa d'una variant xifrada mitjançant SSL anomenada **HTTPS**.

Falta veure com es comunica el navegador amb el servidor web. Com hem dit abans, aquesta comunicació es fa mitjançant el protocol HTTP.

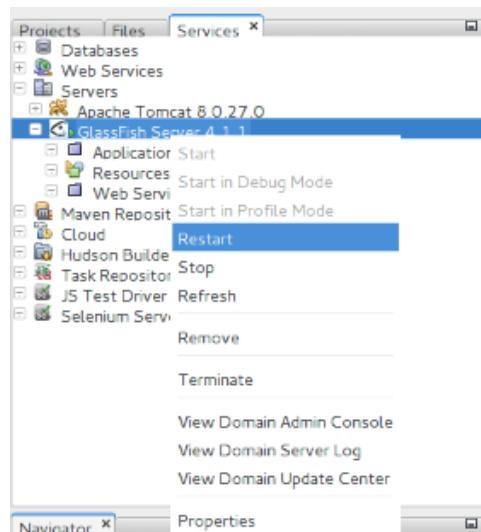
El protocol de transferència d'hipertext o **HTTP** (*HyperText Transfer Protocol*) estableix el protocol per a l'intercanvi de documents d'hipertext i multimèdia al web.

Molt bé, ara activareu la monitorització d'aquest protocol amb el servidor Glassfish. Accediu a les propietats del servidor i activeu la monitorització (vegeu la figura figura 1.9).

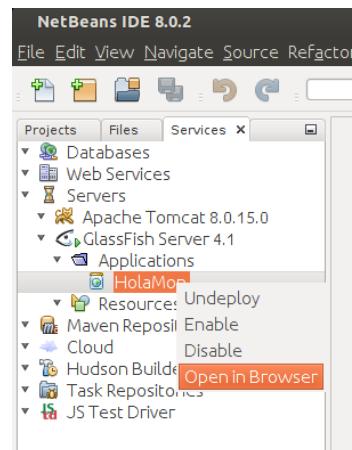
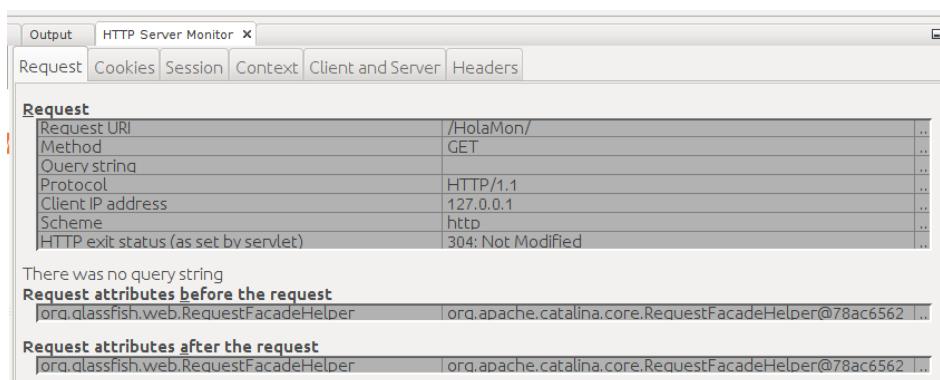
FIGURA 1.9. Activació del monitoratge HTTP



En activar el monitoratge HTTP s'ha de reiniciar el servidor, i finalment s'ha de compilar el projecte un altre cop (vegeu la figura figura 1.10).

FIGURA 1.10. Reiniciar el servidor web

Una vegada heu compilat el projecte podeu obrir-lo al navegador i us apareixerà una finestra al Netbeans amb els missatges HTTP intercanviats (vegeu les figures figura 1.11 i figura 1.12).

FIGURA 1.11. Obrir el recurs web en el navegador**FIGURA 1.12.** Missatge HTTP Get

El propòsit del protocol HTTP és permetre la transferència d'arxius (principalment, en format HTML) entre un navegador (el client) i un servidor web. La comunicació entre el navegador i el servidor es duu a terme en dues etapes:

- El navegador realitza una sol·licitud HTTP.
- El servidor processa la sol·licitud i després envia una resposta HTTP.

Una **sol·licitud HTTP** és un conjunt de línies que el navegador envia al servidor.
Inclou:

- Una línia de sol·licitud: és una línia que especifica el tipus de document sol·licitat, el mètode que s'aplicarà i la versió del protocol utilitzada. La línia està formada per tres elements que han d'estar separats per un espai:
 - el mètode
 - l'adreça URL
 - la versió del protocol utilitzada pel client (en general, HTTP/1.1)
- Els camps de l'encapçalat de sol·licitud: és un conjunt de línies opcionals que permeten aportar informació addicional sobre la sol·licitud i/o el client (navegador, sistema operatiu, etc.). Cadascuna d'aquestes línies està formada per un nom que descriu el tipus d'encapçalat, seguit de dos punts (:) i el valor de l'encapçalat.
- El cos de la sol·licitud: és un conjunt de línies opcionals que han d'estar separades de les línies precedents per una línia en blanc i, per exemple, permeten que s'enviïn dades per una comanda POST durant la transmissió de dades al servidor utilitzant un formulari.

Per tant, una sol·licitud HTTP posseeix la següent sintaxi:

```

1 GET http://ioc.xtec.cat HTTP/1.1
2 Accept : Text/html
3 If-Modified-Since : Friday, 11-December-2015 14:37:11 GMT
4 User-Agent : Mozilla/4.0 (compatible; MSIE 5.0; Windows 95)

```

Les comandes HTTP més importants per fer una sol·licitud són:

- **GET**: s'utilitza per recollir qualsevol tipus d'informació del servidor. S'empra sempre que es prem sobre un enllaç o es tecleja directament a una URL. Com a resultat, el servidor HTTP envia el document corresponent a la URL seleccionada.
- **HEAD**: sol·licita informació sobre un objecte (fitxer): grandària, tipus, data de modificació... És utilitzat pels gestors de *cache* (memòria cau) de pàgines o els servidors *proxy* per conèixer quan és necessari actualitzar la còpia que es manté d'un fitxer.
- **POST**: serveix per enviar informació al servidor, per exemple les dades contingudes en un formulari. El servidor passarà aquesta informació a un procés encarregat del seu tractament (generalment una aplicació CGI/PHP/ASP...). L'operació que es realitza amb la informació proporcionada depèn de la URL utilitzada. S'utilitza sobretot en els formularis.

Una **resposta HTTP** és un conjunt de línies que el servidor envia al navegador. Està constituïda per:

- Una línia d'estat: és una línia que especifica la versió del protocol utilitzada i l'estat de la sol·licitud en procés mitjançant un text explicatiu i un codi. La línia està composta de tres elements que han d'estar separats per un espai:
 - la versió del protocol utilitzada
 - el codi d'estat
 - el significat del codi
- Els camps de l'encapçalat de resposta: és un conjunt de línies opcionals que permeten aportar informació addicional sobre la resposta i/o el servidor. Cadascuna d'aquestes línies està composta d'un nom que qualifica el tipus d'encapçalat, seguit de dos punts (:) i del valor de l'encapçalat.
- El cos de la resposta: conté el document sol·licitat.

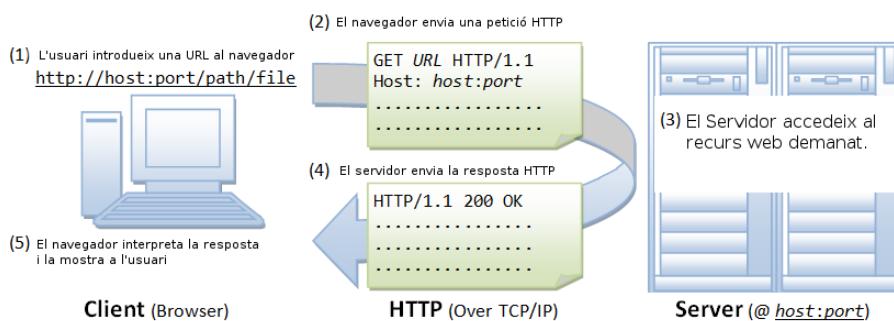
Exemple de resposta d'una petició GET a un servidor web:

```

1 HTTP/1.1 200 OK
2 Server: Microsoft-IIS/5.0\r\n
3 Content-Location: http://www.microsoft.com/default.htm\r\n
4 Date: Fri, 10 Dec 2015 19:33:18 GMT\r\n
5 Content-Type: text/html\r\n
6 Accept-Ranges: bytes\r\n
7 Last-Modified: Thu, 9 Dec 2015 20:27:23 GMT\r\n
8 Content-Length: 26812\r\n
9 <html>
10 ....//pàgina html que es veurà en el navegador
11 </html>
```

La manera de funcionar que des d'un navegador us permet accedir a la informació que existeix en un servidor es diu **arquitectura client-servidor**.

FIGURA 1.13. Arquitectura client-servidor



Penseu diverses aplicacions que utilitzeu diàriament i fan servir l'arquitectura client-servidor.

A les aplicacions client-servidor, al client se'l coneix amb el terme *front-end* o interfície d'usuari. La part client té les següents responsabilitats:

- Interaccionar amb l'usuari.
- Manipular les dades que introduceix l'usuari.
- Enviar les peticions del usuari al servidor.
- Rebre el resultat del processament de les dades enviades al servidor.
- Interpretar la resposta del servidor i mostrar-la a l'usuari.

En canvi, el servidor, també anomenat *back-end*, porta a terme les següents funcions:

- Processar la lògica del programa necessària per donar resposta a la petició del client.
- Realitzar les validacions necessàries a la base de dades.
- Accedir a la base de dades, si fos necessari, per donar una solució als requeriments de la petició enviada pel client.
- Formatar les dades per enviar-les al client.

Per donar a l'usuari una experiència molt més agradable i dinàmica a l'hora d'utilitzar aplicacions client-servidor s'han desenvolupat diferents tecnologies que s'executen o bé al *front-end* o bé al *back-end*.

Proveu el següent codi i digueu si s'executa al *front-end* o al *back-end*:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>TODO supply a title</title>
5          <meta charset="UTF-8">
6          <meta name="viewport" content="width=device-width, initial-scale=1.0">
7          <script src="exemple.js" type="text/javascript"></script>
8          <link href="estils.css" rel="stylesheet" type="text/css"/>
9      </head>
10     <body>
11         <div>HolaMón</div>
12         <button onclick="hola();" value="salutació">salutació</button>
13     </body>
14 </html>
```

Com veieu, en l'HTML anterior utilitzeu dos fitxers addicionals: *estils.css* i *exemple.js*. A continuació teniu el contingut del fitxer *exemple.js*:

```

1  function hola(){
2      alert("hola");
3 }
```

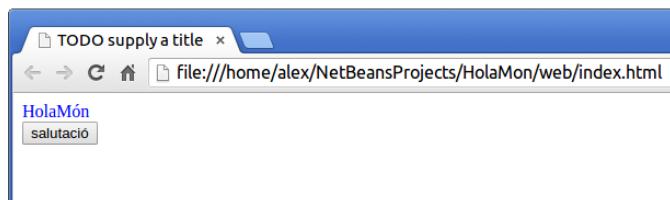
Aquest és el contingut del fitxer “*estils.css*”:

```

1  div{
2      color:blue;
3 }
```

Una manera que teniu per determinar si el codi Javascript i el codi CSS s'executen al servidor o al navegador és executar el codi sense la intervenció del servidor. Igual que heu fet anteriorment, executeu el codi directament fent doble clic al fitxer. La URL que veureu en el navegador és la localització exacta del fitxer en el disc dur (vegeu la figura figura 1.14).

FIGURA 1.14. Execució del projecte directament al navegador



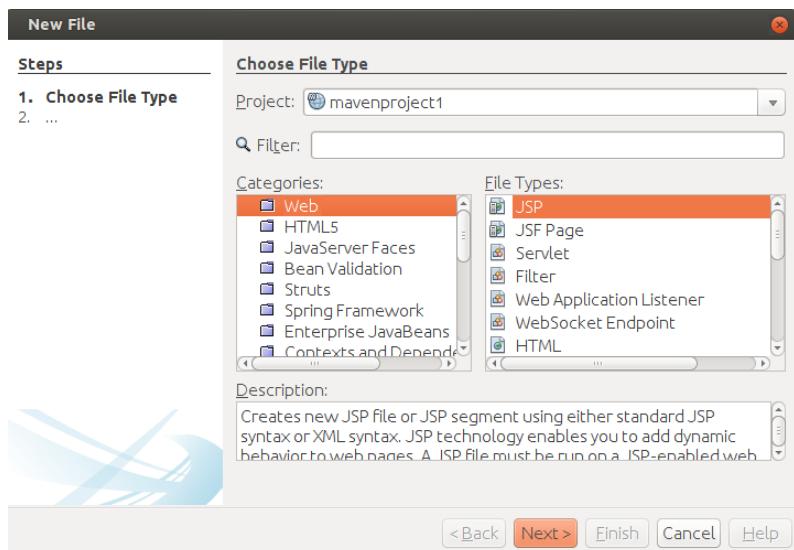
En aquest cas, com podeu comprovar, funciona tot exactament igual que abans. Significa que tant el CSS com el programa en Javascript són tecnologies que s'executen exclusivament en el client. No cal un servidor web perquè funcionin.

Intenteu descobrir quines de les següents tecnologies s'executen al client o al servidor: Java, PHP, CGI, FLASH, JSF, *applets*, CSS, ASP i Javascript.

1.1.3 Pàgines dinàmiques i servidors d'aplicacions

Creareu una pàgina nova, però en comptes de ser de tipus HTML la creareu de tipus JSP (Java Server Pages) i l'anomenareu index.jsp. La creareu utilitzant el mateix projecte de l'apartat anterior mitjançant l'opció *File / New File / Web / JSP* del menú principal (vegeu la figura figura 1.15).

FIGURA 1.15. Nova pàgina JSP



Modifiqueu-la amb el següent codi i digueu si s'executa al *front-end* o al *back-end*:

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
```

```

3  <html>
4      <head>
5          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6          <title>Pàgina JSP</title>
7          <script src="exemple.js" type="text/javascript"></script>
8          <link href="estils.css" rel="stylesheet" type="text/css"/>
9      </head>
10     <body>
11         <div>Hola, Món!</div>
12         <button onclick="hola();" value="salutació">salutació</button>
13         <%>
14             out.println("La teva IP és: " + request.getRemoteAddr());
15         %>
16     </body>
17 </html>

```

Si l'executeu des del programa NetBeans fent servir el servidor web veieu que s'executa tot el codi correctament, fins i tot el codi que hi ha dintre de les etiquetes "<%>" (vegeu la figura figura 1.16).

FIGURA 1.16. Execució de la pàgina index.jsp utilitzant el servidor GlassFish



Però s'ha executat tot en el *front-end* o hi ha algun tros de codi que s'ha executat en el *back-end*?

En comptes d'anar directament al fitxer i fer doble clic, us proposo que mireu el codi font HTML que ha interpretat el navegador. Podeu veure-ho fent clic amb el botó esquerre i després seleccionant del menú l'opció *Visualitza l'origen de la pàgina*, sempre que el ratolí estigui posicionat dintre de la finestra del navegador.

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
5          <title>Pàgina JSP</title>
6          <script src="exemple.js" type="text/javascript"></script>
7          <link href="estils.css" rel="stylesheet" type="text/css"/>
8      </head>
9      <body>
10         <div>Hola, Món!</div>
11         <button onclick="hola();" value="salutació">salutació</button>
12         La teva IP és: 127.0.0.1
13
14     </body>
15 </html>

```

Noteu que tot el codi que hi havia entre les etiquetes "<%>" i "%>" no apareix. En el seu lloc apareix el resultat de la seva execució. Quan el servidor ha volgut respondre a la petició *HTTP GET* del client ha hagut d'executar el codi que hi ha dintre de les etiquetes "<%>" i ha enviat un missatge *HTTP Response* amb el codi HTML substituint les etiquetes "<%>" pel resultat de l'execució.

És a dir, el navegador no ha tingut l'oportunitat de veure que s'havia d'executar codi perquè ni tan sols l'ha rebut. De fet, el servidor web ha fet alguna cosa més del

que feia fins ara. El servidor web s'ha transformat en un **servidor d'aplicacions** on ha hagut de transformar la pàgina web original.

Un **servidor d'aplicacions** és un servidor web i un *framework* de programari on es poden executar aplicacions. És a dir, és un servidor web que permet l'execució d'un programa en el mateix servidor amb les dades proporcionades per una aplicació client.

Com que el servidor ha executat codi JSP, la pàgina HTML ha canviat (l'encerclat per "<% %>"). Qualsevol usuari que es connecti amb el navegador veurà una pàgina diferent. No totalment diferent però sí parcialment, perquè cadascú visualitzarà la seva IP quan accedeixi a aquesta pàgina.

Servidor d'aplicacions Tomcat

NetBeans disposa, a més del servidor GlassFish, del servidor d'aplicacions Tomcat. **Tomcat** és un servidor d'aplicacions d'Apache Software Foundation que executa *servlets* Java i mostra pàgines web que inclouen la codificació Java Server Page. Tomcat és *open source* i està disponible en el lloc web d'Apache.

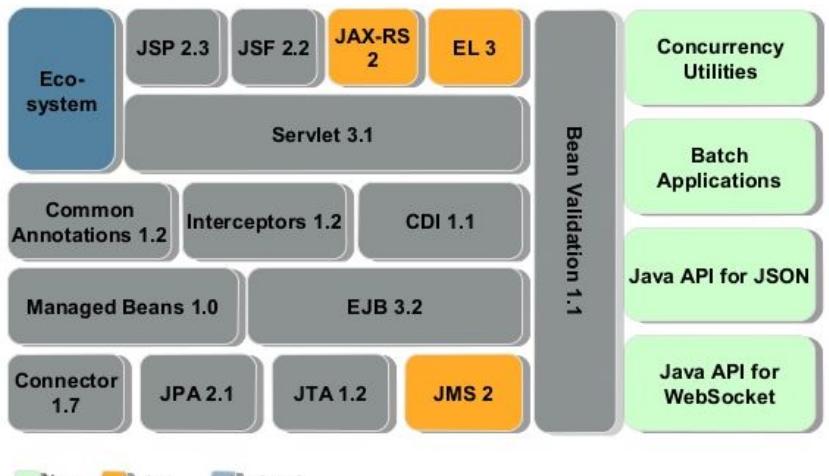
Tomcat implementa un contingut de *servlets* i JSP, que és el que la majoria de les aplicacions web necessiten, però no és un motor de Java EE (Java Enterprise Edition). La seva instal·lació i configuració és molt fàcil, i sovint es pot fer en 10-20 minuts. En ser Tomcat simple i fàcil d'usar, no té diverses característiques importants del Java Enterprise.

Tot i que NetBeans inclou el servidor Tomcat, podeu veure com instal·lar-lo en una màquina Ubuntu en aquest enllaç: <https://help.ubuntu.com/its/serverguide/tomcat.html>.

Tomcat és només un contingut de *servlets*, és a dir, que només implementa l'especificació de *servlets* i JSP. En canvi, **Glassfish** és un servidor de Java EE complet (interpretant tecnologies com EJB, JMS...). Amb Glassfish teniu la implementació de tota la pila Java EE (Java EE stack, vegeu la figura figura 1.17).

FIGURA 1.17. Pila de components Java 7 EE

Java EE 7



Font: Oracle (public)

Proveu el següent exemple:

```

1  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2  <!DOCTYPE html>
3  <html>
4      <head>
5          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6          <title>Pàgina JSP</title>
7          <script src="exemple.js" type="text/javascript"></script>
8          <link href="estils.css" rel="stylesheet" type="text/css"/>
9      </head>
10     <body>
11         <div>Hola, Món!</div>
12         <button onclick="hola();" value="salutació">salutació</button>
13         <p>Avui és <%= new java.util.Date() %> </p>
14     </body>
15 </html>
```

Cada vegada que actualitzeu la pàgina anterior, us retorna la data i l' hora actualitzada. El servidor Glassfish està canviant la pàgina cada vegada que la demanem.

Una **web dinàmica** és aquella que conté aplicacions dins de la mateixa web, atorgant major interactivitat amb el navegant.

Exemples d'aplicacions dinàmiques són enquestes i votacions, fòrums de suport, llibres de visita, enviament de correus electrònics intel·ligents, reserva de productes, comandes *on line*, atenció al client personalitzada...

És important no confondre multimèdia i interactivitat amb pàgines dinàmiques. Una pàgina web estàtica pot ser multimèdia (contenir diversos tipus de vídeos, sons, imatges...) i interactiva a través d'enllaços i hipervincles, sense ser dinàmica per aquestes característiques.

A les pàgines dinàmiques, el contingut sol generar-se al moment de visualitzar-se, poden variar, mentre que en les estàtiques el contingut sol estar predeterminat.

L'important d'aquesta classificació entre dinàmiques i estàtiques és que una pàgina web estàtica la podeu emmagatzemar fàcilment, mentre que en una dinàmica no serà així.

1.1.4 Què s'ha après?

Fins a aquest moment heu après a instal·lar l'entorn de desenvolupament NetBeans. Heu creat el vostre primer projecte amb Maven i heu après el funcionament de la tecnologia de comunicacions client-servidor.

També heu de ser capaços de diferenciar entre un servidor web i un servidor d'aplicacions. S'ha fet una primera aproximació al llenguatge JSP i s'ha explicat la diferència entre una pàgina estàtica i una altre de dinàmica.

Per tal de ser més àgils i autònoms a l'hora de programar, és recomanable fer les activitats d'ampliació on s'explica com depurar el codi des del mateix IDE i com veure la comunicació client-servidor amb eines instal·lades en el navegador web.

Resumint, en aquest apartat s'ha après a:

- Instal·lar i configurar l'entorn de desenvolupament NetBeans.
- Crear un projecte amb Maven.
- Comprendre la comunicació client-servidor, és a dir, el protocol HTTP.
- Configurar el servidor Glassfish.
- Diferenciar una pàgina estàtica d'una dinàmica.

Arribats a aquest punt, hauríeu de ser autònoms a l'hora d'utilitzar l'entorn de desenvolupament NetBeans, així com saber preparar un projecte creat amb Maven. Hauríeu de ser capaços de reconèixer els avantatges de la programació de pàgines web dinàmiques en contraposició a les estàtiques. Finalment, hauríeu de ser capaços d'entendre la comunicació client-servidor amb el protocol HTTP.

1.2 El vostre primer projecte PHP: 'Hola, Món'

Tot i que en aquest mòdul utilitzarà el llenguatge Java com a llenguatge vehicular al llarg de les diferents unitats, és interessant que veieu algunes pinzellades del llenguatge PHP (*PHP Hypertext Preprocessor*), ja que és àmpliament utilitzat en el món empresarial.

En aquest apartat veureu com crear un projecte web PHP amb NetBeans, com instal·lar el programari necessari per posar a punt un servidor PHP i com crear una pàgina molt senzilla.

PHP és un llenguatge de codi obert molt popular, especialment adient per al desenvolupament web, que pot ser afegit directament al codi HTML.

Utilitzareu una màquina virtual amb Ubuntu Server per emprar-la com a servidor web. En aquesta màquina instal·lareu el servidor PHP i utilitzareu el programa Netbeans per crear pàgines web directament al servidor.

1.2.1 Posar a punt el servidor web

L'IDE NetBeans no té un servidor web PHP instal·lat, heu de crear-vos nosaltres un servidor a part. Teniu dues opcions: la primera és instal·lar en la mateixa

màquina on teniu l'IDE el mateix servidor web (amb un XAMPP, per exemple), i la segona és utilitzar una màquina virtual on instal·lar el servidor i fer que l'IDE NetBeans treballi directament amb aquesta.

És molt més real fer-ho de la segona manera. Instal·lareu pas a pas un servidor PHP a una màquina virtual i hi connectareu l'IDE.

Instal·lació del sistema operatiu

En aquest apartat suposeu que teniu instal·lat el programa de màquines virtuals **VirtualBox** i sabeu com crear una màquina nova i com instal·lar un sistema operatiu.

Utilitzareu el servidor Ubuntu Server LTS 14.04. Podeu descarregar-lo de l'adreça: www.ubuntu.com/download/server.

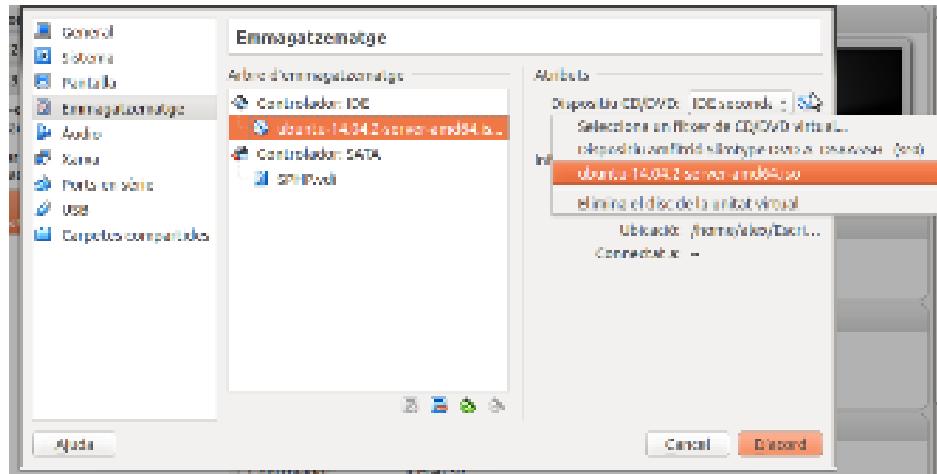
Creareu una màquina virtual nova amb Virtualbox amb les següents característiques (vegeu la figura figura 1.18):

- nom: servidor PHP
- 512 MB de RAM
- 100 GB de disc dur
- xarxa amb adaptador pont

FIGURA 1.18. Configuració servidor PHP amb VBox



Utilitzeu la ISO que heu descarregat dels servidors d'Ubuntu per instal·lar el sistema operatiu. Poseu-la al CD virtual de la màquina “Servidor PHP” (vegeu la figura figura 1.19) i comenceu la instal·lació.

FIGURA 1.19. Selecció de la ISO d'Ubuntu Server al programa Virtualbox

Les característiques per instal·lar el sistema operatiu (vegeu la figura figura 1.20) són les característiques per defecte. No cal instal·lar, en aquest punt, res addicional.

FIGURA 1.20. Pantalla d'instal·lació d'Ubuntu Server

En finalitzar aquest punt s'ha de tenir una màquina virtual amb el sistema operatiu Ubuntu Server instal·lat.

Instal·lació del servidor web

Una vegada teniu preparat el sistema operatiu Ubuntu Server heu de convertir-lo en un servidor web. Per fer-ho heu d'instal·lar una sèrie de paquets, i és molt més còmode si us connecteu via SSH a la màquina virtual; per tant, això és el primer que fareu: instal·lareu el servei SSH.

```
1 sudo apt-get update
2 sudo apt-get install openssh-server
```

L'intèrpret d'ordres segur SSH

SSH (acrònim de *Secure Shell*, 'intèrpret d'ordres segur') és un protocol que permet accedir de manera segura a un ordinador des d'un altre. Permet administrar totalment l'ordinador remot sempre i quan l'usuari tingui els permisos per fer-ho.

A continuació us connectareu via SSH a la màquina virtual. Per poder-vos connectar heu de saber quina adreça IP té el servidor. La podeu esbrinar executant la següent comanda directament al servidor:

```
1 ifconfig
```

Una vegada sabeu l'adreça IP del servidor, podeu utilitzar un terminal de la màquina real i continuar instal·lant els paquets. Per fer la connexió SSH des de la màquina real a la màquina virtual executem:

```
1 ssh usuari_servidor_linux@ip_servidor_linux
2 exemple:
3 ssh alex@192.168.56.101
```

A continuació instal·lareu el servidor Apache2:

```
1 sudo apt-get install apache2
```

I finalment, instal·lareu el mòdul PHP:

```
1 sudo apt-get install php5
2 sudo apt-get install libapache2-mod-php5
```

Només queda configurar els permisos perquè el programa NetBeans pugui escriure a una carpeta propietària de *root*. No configurareu ara els permisos i els usuaris o els grups que tenen permisos per accedir-hi, ja que no és l'objectiu de l'assignatura. Per això, canviareu els permisos perquè tothom pugui entrar a escriure, llegir o executar. En concret, executareu la següent comanda en el terminal d'Ubuntu:

```
1 sudo chmod 777 /var/www/html
```

Una vegada s'ha complert aquest punt ja esteu preparats per crear un nou projecte PHP directament al servidor PHP. Comproveu que ha sortit tot correctament accedint, mitjançant un navegador web des de la màquina real, al servidor PHP. En el vostre cas, per exemple, la IP del servidor és 192.168.56.101 i, com podeu veure a la figura figura 1.21, hi podeu accedir via web.

FIGURA 1.21. Servidor Apache funcionant



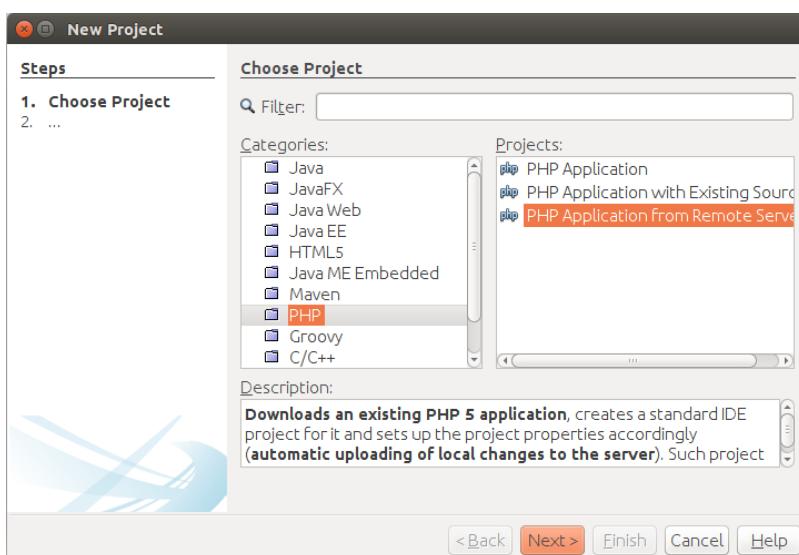
1.2.2 Creació d'un projecte PHP en un servidor remot

Fareu la connexió de l'IDE NetBeans amb el servidor PHP. Com podeu imaginar, funcionaria igual si el servidor tingués una adreça IP pública i vosaltres volguéssiu modificar o crear alguna pàgina PHP.

En crear un nou projecte de tipus PHP, NetBeans dóna tres opcions (vegeu la figura figura 1.22):

- aplicació PHP
- aplicació PHP amb fitxers ja creats
- aplicació PHP des d'un servidor remot

FIGURA 1.22. Nou projecte remot amb PHP

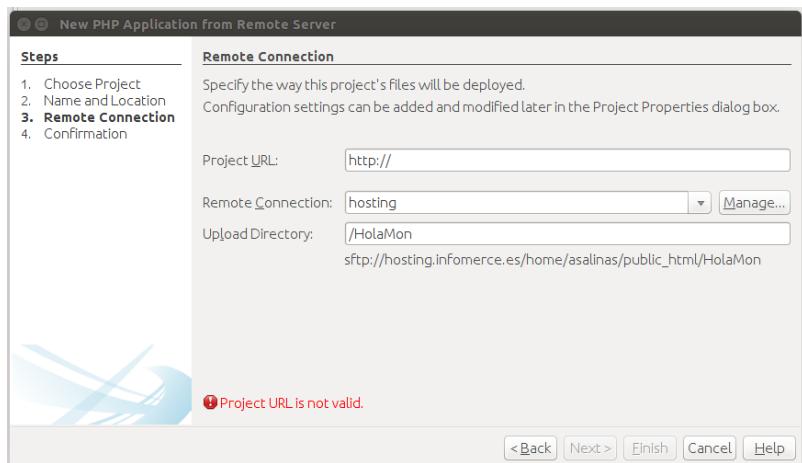


Les dues primeres opcions són vàlides quan el projecte PHP es troba en la mateixa màquina que l'IDE NetBeans. L'única diferència és si es vol crear de zero (primera opció) o ja està creat i es vol afegir o modificar alguna cosa de l'aplicació (segona opció).

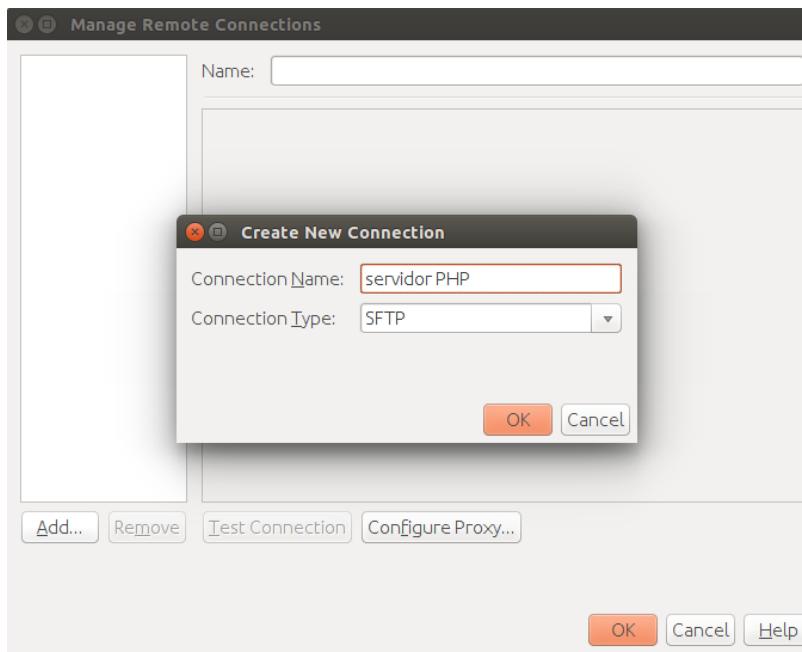
L'última opció és la que fareu servir, aplicació PHP des d'un servidor remot. En el vostre cas, el servidor remot és la màquina virtual (servidor PHP). Una vegada heu escollit aquesta opció, us demana la informació típica d'un projecte:

- nom del projecte (per exemple: *Hola, Món*)
- carpeta amb els fitxers originals
- versió PHP, etc.

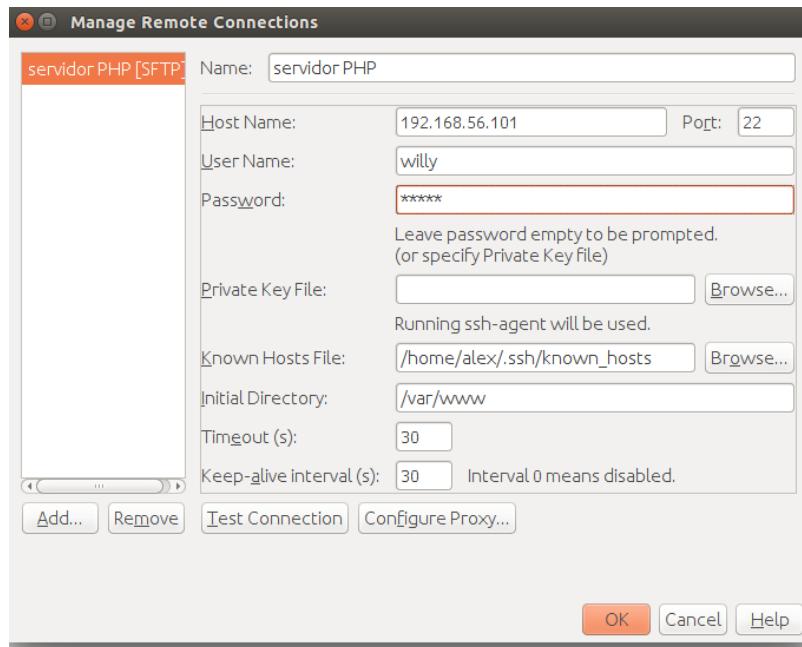
A continuació us demana la configuració per a la connexió amb el servidor remot (vegeu la figura figura 1.23). Haureu de crear primer la connexió per després configurar la carpeta del servidor que voleu utilitzar per copiar els fitxers creats.

FIGURA 1.23. Nova aplicació PHP des d'un servidor remot

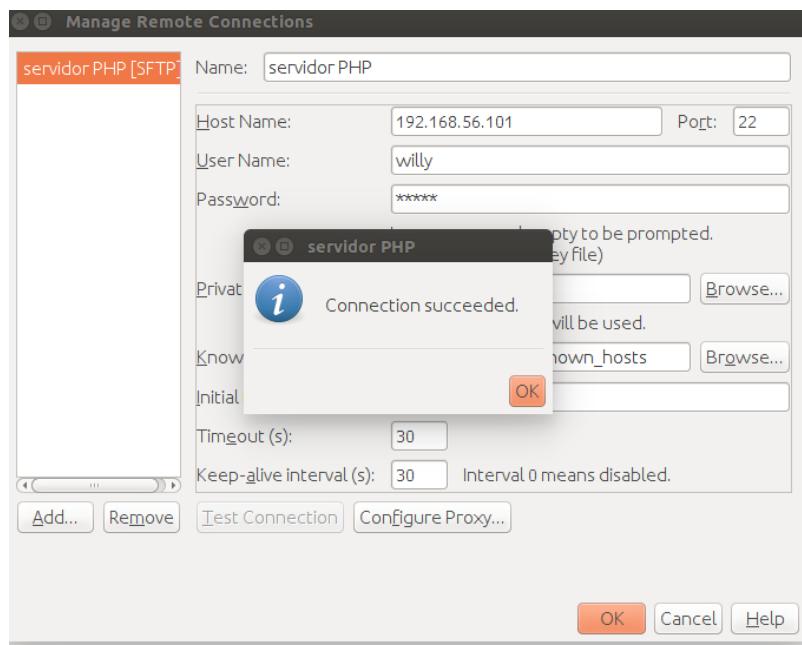
Per crear la connexió anireu a *Administrador (Manage)*. Afegireu una connexió nova de tipus SFTP i li posareu un nom per identificar la connexió, per exemple *Servidor PHP* (vegeu la figura figura 1.24).

FIGURA 1.24. Administrar les connexions als servidors PHP

Una vegada creada la connexió s'ha de configurar afegint la IP del servidor, així com l'usuari del servidor PHP que té permisos per escriure al disc dur. En aquest cas pot ser l'usuari que vau crear en instal·lar el sistema operatiu (vegeu la figura figura 1.25).

FIGURA 1.25. Configurar els paràmetres de la connexió al servidor PHP

Podeu provar la connexió clicant al botó *Test Connection*. Hauria d’aparèixer una finestra semblant a la de la figura figura 1.26.

FIGURA 1.26. Provar la connexió creada

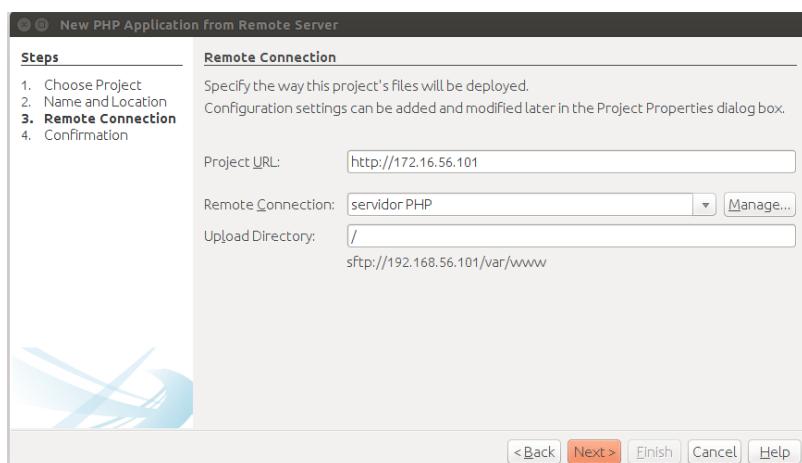
Penseu que si no heu instal·lat el servei SSH al servidor PHP no funcionarà. El servei SSH també instal·la el servei de transferència segura de fitxers SFTP, necessari per establir la connexió entre NetBeans i el servidor PHP.

Finalment, omplireu la finestra que va sortir abans de configurar la connexió amb les dades adients (vegeu la figura figura 1.27):

- URL del projecte: la IP del servidor PHP. Per exemple: <http://192.168.56.101>.

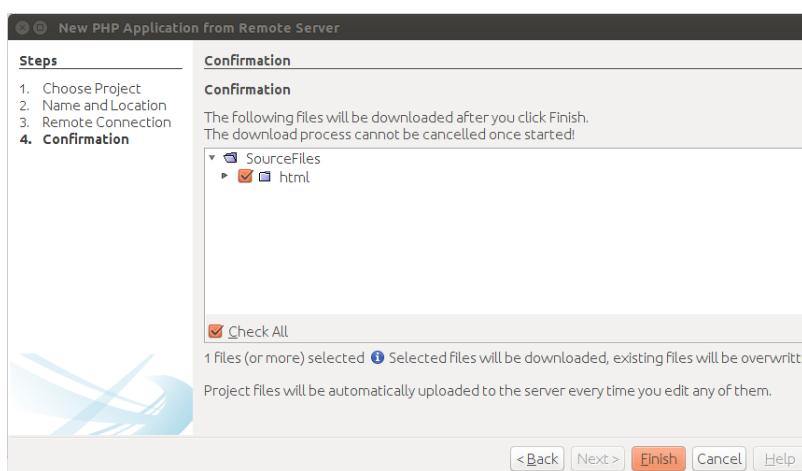
- Connexió remota: seleccioneu la connexió que acabeu de crear (servidor PHP).
- Directori de pujada: directori del servidor PHP on es pujaran els fitxers creats amb el programa NetBeans. Normalment és el mateix directori que utilitza Apache per servir els fitxers web, per exemple `/var/www`. Si poseu `"/"` ja utilitza aquest directori.

FIGURA 1.27. Configuració dels paràmetres de la connexió al servidor PHP



Una vegada heu realitzat tots els passos anteriors, NetBeans detecta que existeix la carpeta HTML i la baixa perquè pugueu crear les pàgines PHP dintre d'aquesta (vegeu la figura figura 1.28).

FIGURA 1.28. Detecció de Netbeans de la carpeta HTML



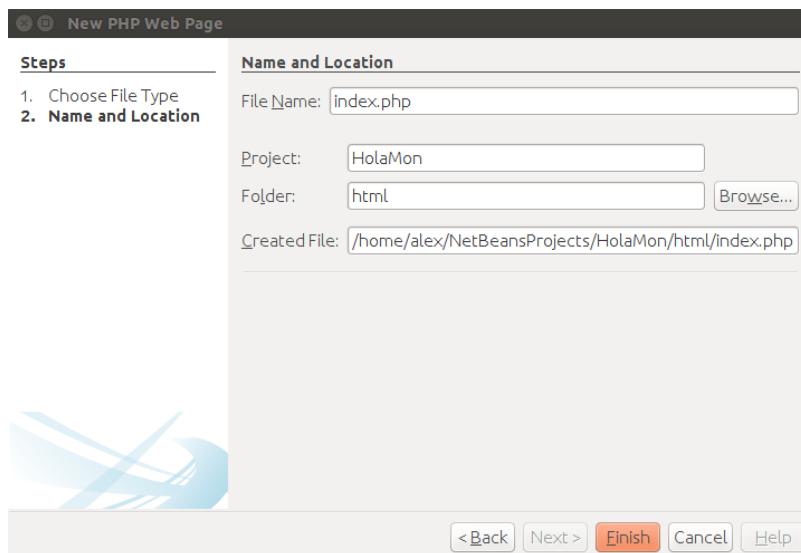
Arribats a aquest punt, ja teniu el projecte preparat per començar a desenvolupar remotament una aplicació PHP.

1.2.3 Exemple 'Hola, Món'

Creareu un nou arxiu dintre de la carpeta HTML del servidor remot. Aquest arxiu ha de ser de tipus PHP, i utilitzareu el menú de NetBeans per crear-lo. Aneu a *File / New File / PHP / PHP Web Page*.

El nom del fitxer serà index.php (vegeu la figura figura 1.29).

FIGURA 1.29. Nou fitxer PHP



Com veieu, a l'hora de crear un fitxer primer es crea a l'ordinador local, és a dir, on està instal·lat l'entorn de desenvolupament. Si us hi fixeu, s'ha creat una còpia del projecte del servidor dintre de la carpeta de projectes locals amb el mateix nom que al servidor. Podeu saber on es troba veient la configuració del paràmetre *Created File* de la figura figura 1.29.

En tenir una còpia al vostre ordinador heu de saber gestionar correctament les versions del projecte, ja que periòdicament haureu de pujar la nova versió al servidor per veure els resultats de la modificació. Com veureu, Netbeans simplifica l'administració de les versions, permetent pujar o baixar del servidor tot el projecte o una part del mateix.

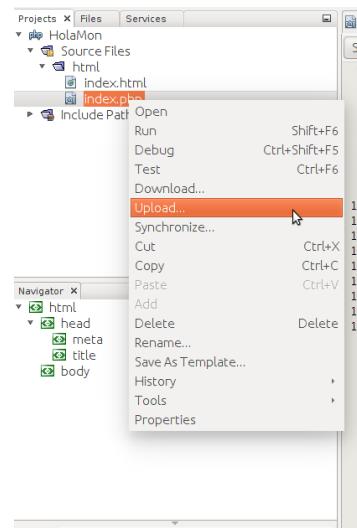
De fet, el servidor no té encara el fitxer creat. Si aneu a la màquina virtual i llisteu els fitxers que hi ha a la carpeta */var/www/html* veureu que només existeix el fitxer *index.html* que té per defecte el servidor Apache. Podeu veure la figura figura 1.30 per comprovar-ho.

FIGURA 1.30. Llistat del fitxers del servidor PHP

```
willy@server:~$  
willy@server:~$  
willy@server:~$ cd /var/www/html/  
willy@server:/var/www/html$ ls  
index.html  
willy@server:/var/www/html$
```

Llavors heu de pujar aquest fitxer al servidor. Si feu clic amb el botó dret del ratolí damunt del fitxer index.php veureu un menú semblant al de la figura figura 1.31. Cliqueu a *Upload*.

FIGURA 1.31. Menú per pujar o baixar un fitxer del servidor PHP



Aquest menú permet executar:

- **Upload:** per pujar un fitxer o una carpeta al servidor.
- **Download:** per baixar una carpeta o fitxer del servidor al PC local.
- **Synchronize:** obre una finestra que us permet escollir si voleu pujar o baixar fitxers del servidor.

Intenteu crear un fitxer nou directament a la màquina virtual i baixe-lo amb NetBeans utilitzant l'opció *Syncronize* o *Download* del menú anterior.

En aquest punt ja heu de tenir el fitxer index.php creat des de NetBeans pujat al servidor PHP. El fitxer index.php ha de ser semblant a aquest:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="UTF-8">
5          <title></title>
6      </head>
7      <body>
8          <?php
9              // put your code here
10             ?>
11         </body>
12     </html>

```

De moment, aquest fitxer no té codi PHP. Només s'han creat dues etiquetes específiques de PHP que permeten escriure codi PHP dintre d'aquestes.

PHP és un llenguatge de programació que coexisteix dintre del llenguatge de marques HTML. L'etiqueta **<?php** és l'etiqueta d'inici d'aquest llenguatge, i

l'etiqueta `?>` és l'etiqueta de tancament. Tot el codi dintre d'aquestes etiquetes no l'interpreta el servidor web Apache2, sinó que el compila i executa el mòdul PHP que s'ha instal·lat a la màquina.

El resultat de l'execució d'aquestes instruccions és el que es mostrerà en comptes del codi que hi ha entre les etiquetes d'inici i fi del llenguatge PHP.

Per exemple, voleu escriure la frase “Hola, Món” des de PHP i que es vegi a la pàgina HTML. El codi que hauríeu d'afegir seria aquest:

```
1 echo "Hola, Món";
```

També podríeu afegir etiquetes de títol (*Heading*), per exemple:

```
1 echo "<h1>Hola, Món</h1>";
```

Com podeu observar, per enviar informació des de PHP a la pàgina HTML s'utilitza la instrucció `echo`, que envia una cadena de text directament a la pàgina HTML en la mateixa posició on es troben les etiquetes PHP dintre de la pàgina.

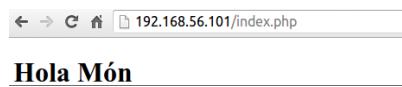
La pàgina “index.php” resultant, després d'afegir la instrucció anterior, seria:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title></title>
6   </head>
7   <body>
8     <?php
9       echo "<h1>Hola, Món</h1>";
10      ?>
11    </body>
12  </html>
```

Guardeu el fitxer anterior i pugeu-lo al servidor.

Com veieu en el navegador web el fitxer index.php? Heu d'accedir a l'adreça del servidor PHP i afegir el recurs que voleu obtenir; en aquest cas, el fitxer index.php. Per exemple, en el vostre cas, en el navegador web haureu d'escriure: 192.168.56.101/index.php (vegeu la figura figura 1.32).

FIGURA 1.32. Execució de la pàgina index.php



Ara fixeu-vos en el codi font que us mostra el navegador web (vegeu la figura figura 1.33). Veieu que no hi ha cap rastre que indiqui que aquest és un fitxer PHP, no apareixen les etiquetes pròpies d'aquest llenguatge. En el seu lloc, apareix el text que heu posat a la instrucció `echo`.

FIGURA 1.33. Codi font de la pàgina index.php

The screenshot shows a browser window with the URL 'view-source:192.168.56.101/index.php'. The page content is a simple PHP script:

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

Observeu que en comunicar-se el navegador amb el servidor PHP aquest no solament ha enviat el codi HTML de la pàgina index.php, sinó que ha hagut de compilar-la i executar-la tal com vam veure amb les pàgines JSP. En aquest cas, el compilador ha estat el mòdul *php5* instal·lat a la màquina virtual.

1.2.4 Què s'ha après?

Heu après que existeixen diferents llenguatges de programació que coexisteixen dintre d'una pàgina web. En aquest cas, heu vist com s'interpreta el llenguatge PHP.

Resumint, heu après a:

- Instal·lar i configurar un servidor PHP.
- Configurar NetBeans per treballar amb un servidor PHP remot.
- Com interacciona un navegador web amb el servidor PHP.
- Crear una pàgina senzilla amb aquest llenguatge.

Ja esteu preparats per començar les activitats proposades en aquest apartat per tal de poder endinsar-vos en el món de la programació amb JSP i PHP.

2. PHP i VDL a Java EE: dades, estructures de control i 'arrays'

En aquesta unitat parlarem dels llenguatges de marques executables per un servidor web identificant-los i escrivint sentències simples per comprovar els seus efectes en la pàgina web resultant. Els llenguatges de marques que s'explicaran seran PHP, JSP i JSTL.

El llenguatge PHP s'explicarà de manera molt breu, ja que el llenguatge vehicular del mòdul serà Java (JEE7 i Spring MVC) i, per tant, el seu llenguatge de marques serà JSP.

Veurem les estructures de control més habituals, el tipus de dades que es poden utilitzar i l'estructura de dades més utilitzada per realitzar iteracions, els *arrays*.

Començarem amb PHP descrivint una persona. Definirem les variables necessàries per guardar la informació i les mostrarem pel navegador. En l'exemple següent utilitzarem els *arrays* per emmagatzemar les dades referents a la gestió d'un hotel i aprendrem a recórrer-los utilitzant les estructures de control més habituals.

Utilitzarem els mateixos exemples per descriure una persona amb JSP i amb JSTL. Així, apreciareu les diferències i les igualtats d'implementar en aquests llenguatges.

En tots els casos s'explicaran els tipus de variable que existeixen, com convertir els valors d'un tipus a un altre i l'àmbit de les variables creades.

En aquest apartat posem les bases de l'aprenentatge d'aplicacions web, i explicarem:

- el llenguatge PHP
- el llenguatge JSP
- el llenguatge JSTL

S'explicarà de cada llenguatge:

- les variables
- els àmbits de les variables
- les estructures de control més habituals
- els tipus de dades i *arrays*

Tots els conceptes que es veuran s'explicaran sempre partint de l'exemple. Quan acabeu aquest apartat estareu preparats per començar a fer les primeres aplicacions web amb PHP i JSP.

No triguem més temps i comencem descrivint una persona amb el llenguatge PHP. Però abans que comenceu aquest apartat comproveu que heu portat a terme la instal·lació de l'entorn PHP proposada a l'apartat anterior. Penseu que per provar els exemples heu de tenir un servidor PHP per executar-los.

2.1 Descrivint una persona amb PHP

En aquest apartat veurem quins tipus de dades existeixen a PHP, com crear variables i quin és el seu àmbit. Aprendrem aquests conceptes en el procés de descriure una persona i intentar emmagatzemar les seves dades.

Començarem creant un nou projecte amb l'IDE Netbeans del tipus PHP/PHP Application. A continuació creem la variable *NomPersona* i li assignem un nom de tipus *string*:

```
1 $nomPersona = "Àlex";
```

Observem que a PHP les variables comencen amb el símbol del dòlar. Sempre, en utilitzar una variable, aquesta ha d'estar precedida per aquest símbol: \$. Les variables a PHP han de complir els següents requisits:

- Han de començar amb el símbol \$.
- El nom de la variable ha de començar per una **lletra** o amb el símbol guíó baix _.
- El nom de la variable no pot començar amb un nombre.
- El nom de la variable només pot contenir nombres, lletres i guions baixos.
- PHP distingeix entre majúscules i minúscules. Les variables \$persona i \$Persona són diferents.

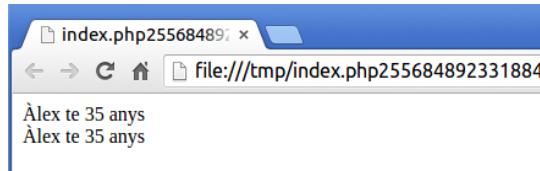
També veiem que en crear la variable \$nomPersona no definim prèviament el seu tipus, ni tan sols el declarem. PHP no suporta la definició explícita de tipus en la declaració de variables, i el tipus d'una variable es determina pel seu context. Aquesta peculiaritat ens permet que una mateixa variable a vegades sigui d'un tipus i unes altres vegades sigui d'un altre.

Executeu el següent codi:

```
1 $nomPersona = "Àlex";
2 $edat = "35";
3 // la variable edat és de tipus string
4 echo "$nomPersona té $edat anys<br>";
5 // Ara la variable edat és de tipus integer:
6 $edat = 34;
7 $edat++;
8 echo "$nomPersona té $edat anys";
```

Observem a la figura figura 2.1 que els dos missatges són idèntics i la variable \$edat primer era de tipus *string* i després ha canviat a ser de tipus *integer*.

FIGURA 2.1. Resultat de l'execució del codi anterior



Ara que ja sabeu crear variables a PHP, intenteu definir el patró dels enginyers informàtics, Ramon Llull. En concret, es vol:

- nom
- cognoms
- edat de tipus numèric
- data de naixement
- telèfon
- adreça postal
- adreça electrònica

Quan hagiu acabat podeu veure la solució:

```

1 $nomPersona = "Ramon";
2 $cognoms = "Llull";
3 $edat = 83; //edat aprox. de la seva mort
4 $data_naixement = strtotime("1232-01-01"); //no se sap el dia exacte
5 $telefon = "935555555";
6 $adrecaPostal = "Ciutat de Mallorca, Convent de Sant Francesc de Palma";
7 $email = "ramon.llull@ioc.cat";

```

A continuació es mostra una altra configuració per a una altra persona:

```

1 $nomPersona = "Clara"; //tipus string
2 $cognoms = "Oswin"; // tipus string
3 $edat = 30; // tipus integer
4 $sou = 30000; //tipus integer
5 $data_naixement = strtotime("1986-03-11"); //timestamp (int)
6 $telefon = "935555555"; // tipus string
7 $adrecaPostal = "Blackpool, England"; //tipus string
8 $email = "oswin@dr.who"; //tipus string
9 $treballa = true; // tipus boolean
10 $alcada = 167.23; // tipus float
11 $convStringData = date('JS F, Y', $data_naixement); //Converteix a string una
    data segons un format donat.

```

Com podeu veure, no cal definir cap tipus. Automàticament, la variable agafa el valor que li hem donat i és quan es defineix el seu tipus. Si executem el codi anterior veiem que no podem veure res per pantalla.

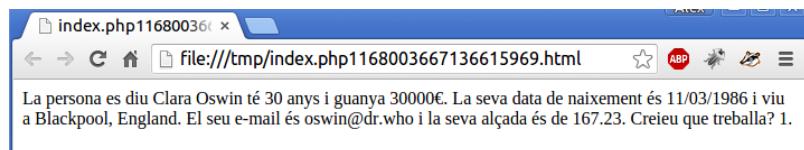
A PHP, en ser un llenguatge que s'executa en el servidor, necessitem una manera de dir que volem enviar una informació a l'usuari. Es fa amb la comanda **echo**.

Llavors, per poder veure per pantalla la informació anterior podem afegir al final la següent instrucció:

```
1 echo "La persona es diu $nomPersona $cognoms té $edat anys i guanya $sou €." .
2 " La seva data de naixement és $convStringData i viu a $adrecaPostal." .
3 " El seu e-mail és $email i la seva alçada és de $alcada. Creieu que treballa?
$treballa.;"
```

Podeu veure el resultat de l'execució del codi anterior en la figura [figura 2.2](#).

FIGURA 2.2. Resultat de l'execució de la descripció d'una persona



En utilitzar la comanda **echo** veieu que sempre intenta convertir les variables a *string*. De fet, concatenar les variables amb els texts que volem donar a l'usuari és molt fàcil. Només s'ha de posar la variable dintre de l'*string* sempre que aquest estigui construït amb dobles cometes (""). També es pot fer amb la forma tradicional concatenant *string* purs amb variables. Per concatenar *strings* s'utilitza el símbol del punt (.).

Si ens fixem, en convertir el booleà *true* a *string* el converteix en el nombre “1”. Si la conversió hagués estat a *integer* el resultat seria 1, però si el valor del booleà hagués estat *false* la seva conversió a *integer* seria un 0 i a *string* seria la cadena buida.

PHP té aquest comportament perquè és un derivat del llenguatge C. Per a tots aquells que sabeu C us serà molt fàcil programar amb PHP, s'assemblen molt.

2.1.1 Conversions de tipus

Tot i que les variables a PHP no tenen tipus, sí que es pot convertir la dada d'un tipus a un altre. Existeixen diferents maneres de fer-ho. Observeu el següent codi:

```
1 $quantitat = "10";
2 $quantitat = (int) $quantitat;
3 $quantitat++;
4 echo $quantitat;
```

El codi anterior funciona correctament. Al final, la variable *\$quantitat* té un valor de 11. Així, amb la conversió a *integer* s'ha pogut convertir el tipus de dada de *string* a *integer* i realitzar una operació matemàtica.

Aquest tipus de conversió es diu que és una **conversió forçada (casting)**. A la taula [taula 2.1](#) podeu veure les diferents conversions forçades que es poden utilitzar.

TAULA 2.1. Conversions de tipus forçades

Instrucció de conversió	Significat
(int), (integer)	conversió a <i>integer</i>
(bool), (boolean)	conversió a <i>boolean</i>
(float), (double), (real)	conversió a <i>float</i>
(string)	conversió a <i>string</i>
(array)	conversió a <i>array</i>
(object)	conversió a <i>object</i>
(unset)	conversió a <i>null</i>

Conversió a booleà

Ara que ja sabem com convertir un tipus de dades ens falta per saber què hem d'esperar. És a dir, quina dada hem d'esperar obtenir després de la conversió. En aquest cas, volem convertir a booleà. Vegeu les conversions a booleà següents:

```

1 var_dump((bool) "");           // bool(false)
2 var_dump((bool) 1);            // bool(true)
3 var_dump((bool) -2);          // bool(true)
4 var_dump((bool) "foo");        // bool(true)
5 var_dump((bool) 2.3e5);        // bool(true)
6 var_dump((bool) array(12));    // bool(true)
7 var_dump((bool) array());      // bool(false)
8 var_dump((bool) "false");      // bool(true)

```

En aquest cas, en comptes d'utilitzar la instrucció *echo* s'ha escollit utilitzar la instrucció ***var_dump***. Aquesta instrucció dóna informació estructurada l'expressió donada incloent-hi el seu tipus i el seu valor.

Sembla, pel codi anterior, que sempre o quasi sempre es converteix a *true* qualsevol cosa. Existeixen molt pocs casos en què un tipus de dades es converteix a *false*, només quan:

- És l'*integer* 0.
- És el *float* 0.0.
- És un *string* buit o un *string* amb el valor “0”.
- Un *array* sense elements.
- El tipus especial *null*.
- Objectes *simpleXML* creats des d'etiquetes buides.

Conversió a 'string'

Hem vist que es pot convertir a *string* utilitzant la conversió forçosa, però també podem utilitzar la funció ***strval()***.

```

1 var_dump(strval(1));          // "1"
2 var_dump(strval(array()));     // "Array"
3 var_dump(strval(-2));         // "-2"
4 var_dump(strval(false));       // ""
5 var_dump(strval(true));        // "1"

```

```

6 var_dump(strval(2.3e5));    // "230000"
7 var_dump(strval(NULL));    // ""

```

Els únics casos remarcables és quan s'intenta convertir un *false*, un *null* o un *array*. En els dos primers casos, la conversió esdevé cadena buida. A l'últim cas, curiosament, ens retorna una cadena amb la paraula *array*.

Tot i així, tots els valors es poden convertir de manera correcta en *string* per després convertir-los novament en el seu tipus. Amb la funció *serialize()* podem emmagatzemar en un *string* un objecte sencer per després tornar-lo a restaurar. La forma de restaurar-lo és utilitzant la funció *unserialize()*.

```

1 $arr = array("a", "e", "i", "o", "u");
2 $serial = serialize($arr);
3 echo($serial);

```

El resultat de l'execució del codi anterior és el següent:

```

1 a:5:{i:0;s:1:"a";i:1;s:1:"e";i:2;s:1:"i";i:3;s:1:"o";i:4;s:1:"u";}

```

Com veieu, aquesta execució ha convertit l'*array* a una cadena, i utilitzant la funció *unserialize()* podem reconstruir l'*array* original. Fem-ho:

```

1 $arrayText = 'a:5:{i:0;s:1:"a";i:1;s:1:"e";i:2;s:1:"i";i:3;s:1:"o";i:4;s:1:"u
2           ";
3 $arrayOriginal = unserialize($arrayText);
echo $arrayOriginal[1]; //e

```

Gràcies a la funció *unserialize()* podem recuperar l'objecte original i tractar les dades normalment.

Conversió a 'integer'

Igual que hem vist amb la conversió de *string*, la conversió a un valor de tipus *integer* es pot realitzar de dues maneres: la primera amb la conversió forçosa utilitzant la paraula reservada (*int*) o (*integer*). La segona, utilitzant la funció *intval()*.

```

1 var_dump(intval("1"));                // 1
2 var_dump(intval(array(3,9,8,6,"w"))); // 1
3 var_dump(intval(array()));            // 0
4 var_dump(intval("-2"));              // -2
5 var_dump(intval(false));             // 0
6 var_dump(intval(true));              // 1
7 var_dump(intval(2.3e5));             // 230000
8 var_dump(intval(NULL));              // 0
9 var_dump(intval("casa"));            // 0

```

Observant el resultat de l'execució del codi anterior notem que si no pot traduir un objecte a un *integer* el resultat de la conversió és el nombre 0. Aquests casos són quan, per exemple, volem convertir l'*string* “casa” o un *array* buit. En canvi, si l'*array* conté algun element es tradueix pel nombre 1.

Conversió a 'array'

Es pot convertir qualsevol dada en un *array*. El que es fa és crear un *array*, i com a únic i primer element la dada que volies convertir en *array*. Exemple:

```
1 $dada = "hola";
2 $arr = (array) $dada;
3 echo $arr[0]; // hola
```

En executar el codi anterior ens retorna la paraula “hola”. Per això, convertir una dada en *array* és el mateix que crear un *array* amb aquesta dada dintre:

```
1 $arr = array("hola");
2 echo $arr[0]; // hola
```

2.1.2 Àmbit de les variables

En els exemples anteriors sempre teníem accés a les variables i podíem veure i manipular el seu valor. No sempre és així. Si tornem a l'exemple del principi, “descriuint una persona”, el podem modificar de la següent manera:

```
1 function calculaEdat($data){
2     $avui = date_create('today');
3     $diferencia = date_diff($avui, $data)->y;
4     return $diferencia;
5 }
6 $nomPersona = "Clara"; //tipus string
7 $cognoms = "Oswin"; // tipus string
8 $data_naixement = date_create("1986-03-11"); //tipus date
9 $edat = calculaEdat($data_naixement);
10 echo $edat;
```

Si executeu el codi anterior veureu que no hi ha cap sorpresa. S'executa perfectament sense error i ens proporciona les dades esperades. Resultat de l'execució:

```
1 29
```

Però podríem fer que la funció *calculaEdat* accedeixi directament a la data de naixement sense haver d'enviar-la per paràmetre? Podríem fer:

```
1 $nomPersona = "Clara"; //tipus string
2 $cognoms = "Oswin"; // tipus string
3 $data_naixement = date_create("1986-03-11"); //tipus date
4 function calculaEdat(){
5     $avui = date_create('today');
6     $diferencia = date_diff($avui, $data_naixement)->y;
7     return $diferencia;
8 }
9 $edat = calculaEdat();
10 echo $edat;
```

En altres paraules, la variable *\$data_naixement* és una variable d'àmbit global? Proveu-ho.

En executar el codi us adoneu que no funciona. No podeu accedir a la variable `$data_naixement` dintre de la funció `calculaEdat()`. Les variables que s'utilitzen dintre d'una funció es troben en un àmbit diferent de les variables que existeixen fora de la funció. Sembla lòic pensar que la variable `$diferencia` no es pot utilitzar fora de la funció:

```

1 $nomPersona = "Clara"; //tipus string
2 $cognoms = "Oswin"; // tipus string
3 $data_naixement = date_create("1986-03-11"); //tipus date
4 function calculaEdat($data){
5     $avui = date_create('today');
6     $diferencia = date_diff($avui, $data )->y;
7     //    return $diferencia;
8 }
9 calculaEdat($data_naixement);
10 echo $diferencia;

```

Efectivament, si ho proveu tampoc funciona. Definitivament, es troben en àmbits diferents.

L'àmbit d'una variable és el context dins de la qual la variable està definida.
La major part de les variables PHP només tenen un àmbit simple.

Àmbit global i àmbit local

No existeixen variables globals? No podem accedir dintre de la funció `calculaEdat` a la data de naixement sense que ens l'enviïn per paràmetre?

Sí que es pot. De fet, la variable `$data_naixement` està declarada en l'**àmbit global**. Totes les variables que es declaren fora de les funcions són variables globals. Podem tenir totes les variables globals que vulguem. En canvi, les variables que es declaren dintre d'una funció són variables **d'àmbit local** a les quals mai podrem accedir fora de la funció (un exemple és la variable `$diferencia`).

Com és que en la variable `$data_naixement`, tot i ser d'àmbit global, no hem pogut accedir al seu valor dintre de la funció `calculaEdat()`?

PHP no sap si vols una variable local amb el mateix nom o vols utilitzar la variable global. Si no informes a PHP que la variable `$data_naixement` és global pensarà que vols una variable local amb aquest nom. Observa el següent codi per veure com utilitzar `$data_naixement` com a variable global:

```

1 function calculaEdat(){
2     global $data_naixement;
3     $avui = date_create('today');
4     $diferencia = date_diff($avui, $data_naixement )->y;
5     return $diferencia;
6 }
7 $nomPersona = "Clara"; //tipus string
8 $cognoms = "Oswin"; // tipus string
9 $data_naixement = date_create("1986-03-11"); //tipus date
10 $edat = calculaEdat();
11 echo $edat;

```

S'executa perfectament sense error i ens proporciona les dades esperades. Resultat de l'execució:

 1 29

Dintre de la funció `calculaEdat()` hem hagut de declarar la variable `$data_naixement` com a global. Hem utilitzat la paraula reservada **global**.

També podem utilitzar l'*array* associatiu **GLOBALS** per accedir a les variables globals i, d'aquesta manera, ens estalviem haver de declarar-la *global*.

```
1 function calculaEdat(){
2     $avui = date_create('today');
3     $diferencia = date_diff($avui, $GLOBALS["data_naixement"] )->y;
4     return $diferencia;
5 }
6 $nomPersona = "Clara"; //tipus string
7 $cognoms = "Oswin"; // tipus string
8 $data_naixement = date_create("1986-03-11"); //tipus date
9 $edat = calculaEdat();
10 echo $edat;
```

Funciona exactament igual que abans, el resultat de l'execució és:

 1 29

Ara veiem una peculiaritat dintre de l'àmbit local. Normalment, les variables perdren el seu valor una vegada la funció s'ha executat, o no?

```
1 function suma(){
2     static $num;
3     $num++;
4     return $num;
5 }
6 echo suma();
7 echo suma();
8 echo suma();
```

El resultat de l'execució del codi anterior és:

 1 1
 2 2
 3 3

La paraula reservada **static** permet guardar el valor d'una variable d'àmbit local dintre de la funció. Cada vegada que s'executi la funció, la variable tindrà l'últim valor que havia tingut a l'execució anterior.

2.2 Gestionant un hotel amb PHP

En aquest apartat veurem el tipus de dades *array* i les estructures de control necessàries per tractar amb aquest tipus de dades. Aprendrem aquests conceptes gestionant, d'una manera molt senzilla, alguns aspectes de la gestió d'un hotel.

2.2.1 Gestió de les taules del restaurant del hotel

Començarem amb la gestió de les taules del restaurant. Volem saber quants comensals hi ha en cadascuna de les taules del restaurant. Suposeu que tenim 10 taules i que en cadascuna de les taules hi pot haver fins a 5 comensals. Aneu pensant quins passos haurem de realitzar per aconseguir-ho.

Si només tinguéssim una taula en el restaurant escriuríem un codi com aquest:

```
1 $comensals = rand(0,5);
2 echo "A la taula hi ha $comensals comensals";
```

La funció *rand(min, max)* ens dóna un nombre pseudoaleatori entre el nombre mínim i el nombre màxim definits en la crida de la funció. Un possible resultat de l'execució del codi anterior seria:

```
1 A la taula hi ha 3 comensals.
```

Imagineu que la funció aleatòria ens retorna el valor 0. No seria millor que el programa ens digués que “la taula està buida”, en comptes de “a la taula hi ha 0 comensals”? Modifiquem el codi anterior per afegir aquesta millora:

```
1 $comensals = rand(0,5);
2 if ($comensals == 0){
3     echo "La taula està buida";
4 }
5 else{
6     echo "A la taula hi ha $comensals comensals";
7 }
```

L'estructura de control *if* és una de les estructures més importants. Permet l'execució condicional d'un tros de codi. Funciona de la següent manera:

- L'expressió que hi ha entre parèntesis, després de la paraula clau *if*, s'avalua i el resultat ha de ser *true* o *false*.
- Si s'avalua com a *true* s'executarà el tros de codi immediatament posterior.
- En canvi, si l'expressió s'avalua com a *false* s'executarà el tros de codi que hi ha després de la paraula reservada *else*.
- En ser una estructura condicional, en cap cas s'executaran els dos trossos de codi a la vegada.

Per exemple, si la variable *\$comensals* tingués el valor *1, 2, 3, 4 o 5*, el resultat de l'execució seria:

```
1 A la taula hi ha 1 comensals // quan $comensals = 1
```

En canvi, si la variable *\$comensals* tingués el valor *0*, el resultat de l'execució seria:

1 La taula està buida

Molt bé, doncs ara volem afegir el condicional de quan la taula està plena. No volem que ens digui que hi ha 5 comensals, si no que volem que ens digui la frase: “La taula està plena”.

```
1 $comensals = rand(0,5);
2 if ($comensals == 0){
3     echo "La taula està buida";
4 }
5 elseif ($comensals == 5){
6     echo "La taula està plena";
7 }
8 else {
9     echo "A la taula hi ha $comensals comensals";
10 }
```

En aquest cas afegim l’estructura *elseif(expressió)*. Com afecta, en l’execució del codi anterior, haver afegit aquesta estructura?

- L’expressió que hi ha entre parèntesis, després de la paraula clau *if*, s’avalua i el resultat ha de ser *true* o *false*.
- Si s’avalua com a *true* s’executarà el tros de codi immediatament posterior.
- En canvi, si l’expressió s’avalua com a *false* s’avaluarà l’expressió de l’estructura *elseif*. Si aquesta s’avalua com a *true* s’executarà el tros de codi immediatament posterior.
- Si no, si s’avalua com a *false* s’executarà el tros de codi que hi ha després de la paraula reservada *else*.
- En ser una estructura condicional, en cap cas s’executaran els tres trossos de codi a la vegada.

Us recomanem que feu diverses proves forçant la variable *\$comensals* perquè executi cadascun dels trossos anteriors. Ho podeu fer actualitzant la pàgina (F5) per canviar el valor donat de la funció *random (rand)*.

Ha arribat el moment de començar a ampliar el restaurant. Amb una única taula no fareu negoci. Tornem al problema que es va plantejar al principi: voleu saber quants comensals hi ha en cadascuna de les taules del restaurant. Supposeu que tenim 10 taules i en cadascuna de les taules hi pot haver fins a 5 comensals.

Si heu estat pensant quina és la millor manera de guardar les dades segur que heu arribat a la conclusió que un *array* ens facilitaria la gestió de les dades. Podem tenir ordenades les taules i podem guardar el nombre de comensals en cadascuna de les seves posicions.

```
1 $taules = array();
2 $taules[0] = rand(0,5);
3 $taules[1] = rand(0,5);
4 $taules[2] = rand(0,5);
5 $taules[3] = rand(0,5);
6 $taules[4] = rand(0,5);
```

```

7 $taules[5] = rand(0,5);
8 $taules[6] = rand(0,5);
9 $taules[7] = rand(0,5);
10 $taules[8] = rand(0,5);
11 $taules[9] = rand(0,5);

```

En el codi anterior, veiem com guardar el nombre de comensals en cadascuna de les taules. Els *arrays* en PHP no cal inicialitzar-los amb el nombre de posicions que ha de tenir (la seva mida). Cada vegada que accedim a una posició de l'*array*, aquesta es crea i s'afegeix la dada. També ho podríem haver codificat de la següent manera:

```

1 $taules = array();
2 array_push($taules, rand(0,5));
3 array_push($taules, rand(0,5));
4 array_push($taules, rand(0,5));
5 array_push($taules, rand(0,5));
6 array_push($taules, rand(0,5));
7 array_push($taules, rand(0,5));
8 array_push($taules, rand(0,5));
9 array_push($taules, rand(0,5));
10 array_push($taules, rand(0,5));
11 array_push($taules, rand(0,5));

```

En aquest cas utilitzem la funció ***array_push***, que afegeix una dada al final de l'*array*. Va creant noves posicions a mesura que es necessiti afegir dades noves.

Tot i que els codis anteriors són correctes i inicialitzen l'*array*, sembla que estem repetint molt de codi, no? I si en comptes de 10 taules tinguéssim 100? Sembla estrany haver de repetir una línia de codi 100 vegades. Observeu la següent estructura de control:

```

1 $taules = array();
2 for($numTaula=0;$numTaula < 10 ; $numTaula++){
3     $taules[$numTaula] = rand(0,5);
4 }
5 echo $taules[4];

```

En aquest cas estem repetint una instrucció deu vegades, però cada vegada que es repeteix canvia la variable *\$numTaula*. A cada volta la variable canvia el seu valor, segons el tercer paràmetre de l'estruatura de control, *\$numTaula++*.

Utilitzem aquesta variable per canviar de posició en l'*array* *\$taules* i d'aquesta manera podem afegir els comensals a la nova taula.

L'estruatura de control ***for*** té la següent estructura:

```

1 for (inicialitzar comptador; comparació per saber quan parar; incrementar
      comptador) {
2     codi a executar;
3 }

```

Aquesta estructura és útil quan saps quantes vegades s'han d'executar les instruccions. En aquest cas, com que tenim 10 taules s'ha d'executar 10 vegades. De fet, és l'estruatura típica del bucle *for*. Però a PHP hi ha un tipus de bucle *for* que és exclusiu per a *arrays*.

Utilitzarem un *array* associatiu en què en comptes de tenir un nombre per accedir a una posició de l'*array* utilitzarem un *string*. Observeu:

```

1 //creació de l'array
2 $taules = array();
3
4 //Omplir l'array associatiu:
5 for($numTaula=0;$numTaula < 10 ; $numTaula++){
6     $taules["taula ".$numTaula] = rand(0,5);
7 }
8
9 //visualització del contingut de l'array
10 foreach($taules as $posicio => $comensals){
11     echo "La $posicio té $comensals comensals\n<br>";
12 }
```

Una possible sortida de l'execució del codi anterior:

```

1 La taula 0 té 4 comensals
2 La taula 1 té 1 comensals
3 La taula 2 té 4 comensals
4 La taula 3 té 1 comensals
5 La taula 4 té 5 comensals
6 La taula 5 té 3 comensals
7 La taula 6 té 3 comensals
8 La taula 7 té 5 comensals
9 La taula 8 té 1 comensals
10 La taula 9 té 5 comensals
```

Amb l'estruatura de control *foreach* podem accedir a la clau (*\$posicio*) i al valor(*\$comensals*). Aquests valors s'omplen de manera automàtica i a cada volta ens dóna la següent posició i el seu valor corresponent: *\$comensals = \$taules[\$posicio];*.

L'estruatura de control *foreach* té la següent estructura:

```

1 foreach (expresio_array as $clau => $valor){
2     sentencies
3 }
```

Doncs ara, per acabar la gestió del restaurant haurem d'unir les estructures condicionals per saber quan una taula està plena o buida amb les estructures de control iteratives.

```

1 //creació de l'array
2 $taules = array();
3
4 //Omplir l'array associatiu:
5 for($numTaula=0;$numTaula < 10 ; $numTaula++){
6     $taules["taula ".$numTaula] = rand(0,5);
7 }
8
9 //visualització del contingut de l'array
10 foreach($taules as $posicio => $comensals){
11     if ($comensals == 0){
12         echo "La $posicio està buida\n<br>";
13     }
14     elseif ($comensals == 5){
15         echo "La $posicio està plena\n<br>";
16     }
17     else {
18         echo "A la $posicio hi ha $comensals comensals\n<br>";
19     }
}
```

20 }

Un possible resultat del codi anterior és el següent:

```

1 A la taula 0 hi ha 2 comensals
2 A la taula 1 hi ha 3 comensals
3 A la taula 2 hi ha 1 comensals
4 A la taula 3 hi ha 3 comensals
5 La taula 4 està buida
6 A la taula 5 hi ha 2 comensals
7 La taula 6 està plena
8 La taula 7 està plena
9 La taula 8 està plena
10 La taula 9 està buida

```

2.2.2 Gestió de les habitacions de l'hotel

En aquest apartat es veurà el tractament d'informació amb *arrays* de més d'una dimensió. Aprendreu com crear *arrays* multidimensionals, com modificar els valors i com cercar-los.

Es tractarà de programar la gestió d'habitacions d'un hotel. Imagineu un hotel amb 5 plantes i 10 habitacions en cadascuna de les plantes. Es vol guardar el nombre de clients que hi ha en cada habitació. Com a màxim hi pot haver 4 clients per habitació.

Començarem creant un *array* de dues dimensions:

```

1 define("NUM_PLANTES", 5);
2 define("NUM_HAB", 10);
3 $habitaciones = array();
4 for($pis=0; $pis < NUM_PLANTES; $pis++){
5     $habitaciones[$pis] = array();
6     for($porta=0; $porta < NUM_HAB; $porta++){
7         $habitaciones[$pis][$porta] = "";
8     }
9 }

```

Veieu que s'utilitza la funció ***define***, que defineix una constant en el programa.

Una **constant** és un identificador que s'utilitza per expressar un valor simple, que no variarà en el transcurs del programa. Per convenció, els identificadors de les constants sempre se solen declarar en majúscules.

Per utilitzar la constant en el codi només s'ha d'utilitzar el nom definit prèviament a la funció ***define***.

El codi anterior genera una estructura com aquesta:

```

1 Array
2 (
3     [0] => Array
4         (

```

```
5      [0] =>
6      [1] =>
7      [2] =>
8      [3] =>
9      [4] =>
10     [5] =>
11     [6] =>
12     [7] =>
13     [8] =>
14     [9] =>
15   )
16
17 [1] => Array
18   (
19     [0] =>
20     [1] =>
21     [2] =>
22     [3] =>
23     [4] =>
24     [5] =>
25     [6] =>
26     [7] =>
27     [8] =>
28     [9] =>
29   )
30
31 [2] => Array
32   (
33     [0] =>
34     [1] =>
35     [2] =>
36     [3] =>
37     [4] =>
38     [5] =>
39     [6] =>
40     [7] =>
41     [8] =>
42     [9] =>
43   )
44
45 [3] => Array
46   (
47     [0] =>
48     [1] =>
49     [2] =>
50     [3] =>
51     [4] =>
52     [5] =>
53     [6] =>
54     [7] =>
55     [8] =>
56     [9] =>
57   )
58
59 [4] => Array
60   (
61     [0] =>
62     [1] =>
63     [2] =>
64     [3] =>
65     [4] =>
66     [5] =>
67     [6] =>
68     [7] =>
69     [8] =>
70     [9] =>
71   )
72
73 )
```

Per crear un *array* de dues dimensions (matriu bidimensional) a PHP hem de crear un *array d'arrays*, és a dir, un *array* en què cadascuna de les seves posicions és un altre *array*. En aquest cas, amb el primer índex de l'*array* *\$habitacions* accedim al pis, i amb el segon índex accedim a l'habitació d'aquell pis.

Per exemple, si accedim a la posició de l'*array* *\$habitacions[0][4]* estem accedint al pis 0 (planta baixa) habitació 4.

Doncs bé, igual que abans amb el restaurant, ara ens falta inicialitzar l'*array* amb el nombre de clients per habitació.

```

1 define("NUM_PLANTES", 5);
2 define("NUM_HAB", 10);
3 define("MAX_CLIENTS", 4);
4
5 $habitacions = array();
6 for($pis=0; $pis < NUM_PLANTES; $pis++){
7     $habitacions[$pis] = array();
8     for($porta=0; $porta < NUM_HAB; $porta++){
9         $habitacions[$pis][$porta] = rand(0,MAX_CLIENTS);
10    }
11 }
```

Només calia afegir la constant *MAX_CLIENTS*, que indica el nombre màxim de persones que hi pot haver en una habitació i la crida a la funció *random* (*rand()*) perquè de manera aleatòria afageixi persones a les habitacions.

Ara volem un llistat amb l'ocupació de les habitacions de l'hotel. Aquest llistat s'assemblarà molt al que hem fet abans amb les taules del restaurant.

Aquells que ja teniu una idea de com implementar-ho aneu pensant com faríeu per saber si hi ha alguna habitació lliure.

Codi per obtenir el llistat amb el nombre de persones per habitació:

```

1 //Llistat del nombre de persones per habitació
2
3 //Crear l'estructura inicial
4 define("NUM_PLANTES", 5);
5 define("NUM_HAB", 10);
6 define("MAX_CLIENTS", 4);
7 $habitacions = array();
8 for($pis=0; $pis < NUM_PLANTES; $pis++){
9     $habitacions[$pis] = array();
10    for($porta=0; $porta < NUM_HAB; $porta++){
11        $habitacions[$pis][$porta] = rand(0,MAX_CLIENTS);
12    }
13 }
14
15 //recórrer l'estructura per obtenir la informació:
16 for($pis=0; $pis < NUM_PLANTES; $pis++){
17     for($porta=0; $porta < NUM_HAB; $porta++){
18         switch ($habitacions[$pis][$porta]){
19             case 0:
20                 echo "La habitació $porta de la planta $pis està buida.\n";
21                 break;
22             case 4:
23                 echo "La habitació $porta de la planta $pis està plena.\n";
24                 break;
25             default :
26                 echo "A la habitació $porta de la planta $pis hi ha ".
27                     $habitacions[$pis][$porta]. " persones.\n";
28         }
29     }
30 }
```

```
28     }
29 }
```

El resultat de l'execució del codi anterior és el següent:

```
1 A l'habitació 0 de la planta 0 hi ha 1 persones.
2 L'habitació 1 de la planta 0 està buida.
3 L'habitació 2 de la planta 0 està plena.
4 L'habitació 3 de la planta 0 està buida.
5 A l'habitació 4 de la planta 0 hi ha 3 persones.
6 L'habitació 5 de la planta 0 està buida.
7 L'habitació 6 de la planta 0 està plena.
8 A l'habitació 7 de la planta 0 hi ha 1 persones.
9 A l'habitació 8 de la planta 0 hi ha 3 persones.
10 L'habitació 9 de la planta 0 està buida.
11 L'habitació 0 de la planta 1 està plena.
12 A l'habitació 1 de la planta 1 hi ha 3 persones.
13 A l'habitació 2 de la planta 1 hi ha 2 persones.
14 L'habitació 3 de la planta 1 està buida.
15 A l'habitació 4 de la planta 1 hi ha 1 persones.
16 A l'habitació 5 de la planta 1 hi ha 2 persones.
17 L'habitació 6 de la planta 1 està plena.
18 A l'habitació 7 de la planta 1 hi ha 3 persones.
19 A l'habitació 8 de la planta 1 hi ha 3 persones.
20 L'habitació 9 de la planta 1 està plena.
21 L'habitació 0 de la planta 2 està plena.
22 L'habitació 1 de la planta 2 està buida.
23 A l'habitació 2 de la planta 2 hi ha 1 persones.
24 A l'habitació 3 de la planta 2 hi ha 2 persones.
25 L'habitació 4 de la planta 2 està plena.
26 L'habitació 5 de la planta 2 està plena.
27 A l'habitació 6 de la planta 2 hi ha 2 persones.
28 L'habitació 7 de la planta 2 està plena.
29 A l'habitació 8 de la planta 2 hi ha 2 persones.
30 A l'habitació 9 de la planta 2 hi ha 2 persones.
31 L'habitació 0 de la planta 3 està buida.
32 L'habitació 1 de la planta 3 està plena.
33 A l'habitació 2 de la planta 3 hi ha 2 persones.
34 L'habitació 3 de la planta 3 està buida.
35 L'habitació 4 de la planta 3 està buida.
36 L'habitació 5 de la planta 3 està buida.
37 L'habitació 6 de la planta 3 està buida.
38 L'habitació 7 de la planta 3 està plena.
39 A l'habitació 8 de la planta 3 hi ha 2 persones.
40 L'habitació 9 de la planta 3 està plena.
41 L'habitació 0 de la planta 4 està plena.
42 A l'habitació 1 de la planta 4 hi ha 1 persones.
43 A l'habitació 2 de la planta 4 hi ha 2 persones.
44 A l'habitació 3 de la planta 4 hi ha 2 persones.
45 A l'habitació 4 de la planta 4 hi ha 2 persones.
46 A l'habitació 5 de la planta 4 hi ha 3 persones.
47 L'habitació 6 de la planta 4 està plena.
48 A l'habitació 7 de la planta 4 hi ha 2 persones.
49 A l'habitació 8 de la planta 4 hi ha 2 persones.
50 A l'habitació 9 de la planta 4 hi ha 3 persones.
```

Com podeu observar, l'estrucció per recórrer la matriu bidimensional i l'estrucció per crear-la és la mateixa. S'ha de recórrer cada *array* que s'ha creat a cada posició de l'*array* principal. Una vegada hem accedit a una habitació d'una planta s'ha de comprovar el nombre de persones que hi ha.

Aquesta vegada, per fer-ho una mica diferent, s'ha utilitzat l'estrucció ***switch***. Aquesta estructura de control és similar a una sèrie de sentències *if*. En moltes ocasions, és possible que es vulgui comparar la mateixa variable (o expressió) amb molts valors diferents i executar una part del codi dependent de quin valor és igual.

Exemple de dues maneres d'escriure el mateix: la primera amb les estructures de control *if* i la segona amb l'estructura de control *switch*:

```

1  if ($i == 0) {
2      echo "i és igual a 0";
3  } elseif ($i == 1) {
4      echo "i és igual a 1";
5  } elseif ($i == 2) {
6      echo "i és igual a 2";
7  }
8  switch ($i) {
9      case 0:
10         echo "i és igual a 0";
11         break;
12     case 1:
13         echo "i és igual a 1";
14         break;
15     case 2:
16         echo "i és igual a 2";
17         break;
18 }
```

Ha arribat l'hora de solucionar el repte plantejat anteriorment. Volem saber si existeix una habitació lliure. Volem recórrer la matriu bidimensional, però quan hagi trobat la primera habitació lliure ha de terminar el programa. L'estructura de control que ens permet parar la cerca una vegada hem trobat la dada és la següent:

```

1  while (expr){
2      sentencies
3  }
```

Per realitzar la cerca demandada primer inicialitzarem la matriu i després, amb aquesta estructura de control, buscarem la primera habitació on hi hagi 0 persones:

```

1 //Cercar si hi ha habitacions lliures
2 //Crear l'estructura inicial
3 define("NUM_PLANTES", 5);
4 define("NUM_HAB", 10);
5 define("MAX_CLIENTS", 4);
6 $habitaciones = array();
7 for($pis=0; $pis < NUM_PLANTES; $pis++){
8     $habitaciones[$pis] = array();
9     for($porta=0; $porta < NUM_HAB; $porta++){
10         $habitaciones[$pis][$porta] = rand(0,MAX_CLIENTS);
11     }
12 }
13 //recórrer l'estructura per obtenir la informació:
14 $pis=0;
15 $porta=0;
16 $trobat = false;
17 while (!$trobat && $pis < NUM_PLANTES){
18     if($habitaciones[$pis][$porta] === 0){
19         $trobat = true;
20     }
21     if($porta == NUM_HAB - 1){
22         $porta = 0;
23         $pis++;
24     }
25     else{
26         $porta++;
27     }
28 }
29 echo ($trobat)? "Almenys hi ha una habitació lliure." : "No existeixen
    habitacions lliures.;"
```

Per recórrer la matriu tenim dos índexs: `$pis` i `$porta`. Amb el primer índex accedim a la planta de l'hotel, i amb el segon a l'habitació d'aquella planta. Primer hem d'inicialitzar els índexs:

```
1 $pis = 0;
2 $porta = 0;
```

Una vegada inicialitzats ja podem recórrer la matriu. Com? Doncs pis a pis. Hem de recórrer totes les habitacions del primer pis (`$porta++`), i quan les haguem recorregut hem d'anar al següent pis. Anar al següent pis significa que hem arribat a l'última porta, llavors inicialitzem el número de porta (`$porta = 0;`) i a l'índex `$pis` sumar-hi un més.

```
1 if($porta == NUM_HAB -1){
2     $porta = 0;
3     $pis++;
4 }
5 else{
6     $porta++;
7 }
```

Però hem dit que hem de cercar la primera habitació on no hi hagi cap persona. Quan trobem una que compleixi la condició hem d'avalar, posant una variable a *true*, que ja hem trobat el que estàvem cercant:

```
1 if($habitaciones[$pis][$porta] === 0){
2     $trobat = true;
3 }
```

En aquest cas, utilitzem una comparació amb tres símbols `==`. Els tres iguals significa que el valor ha de ser el mateix, com sempre, però a més a més ha de ser del mateix tipus. En aquest cas, ha de ser un 0 i de tipus enter, ja que al costat dret de la comparació utilitzem directament el símbol de tipus enter amb valor 0.

Si trobem una habitació amb 0 persones la variable `$trobat` canvia el seu valor a *true*. Fent aquest canvi, la condició per continuar dintre del bucle esdevé *false* i no tornarà a executar cap sentència de dintre del bucle.

```
1 $trobat = false;
2 while (! $trobat && $pis < NUM_PLANTES){
3     if($habitaciones[$pis][$porta] === 0){
4         $trobat = true;
5     }
6     ....
7 }
```

El bucle acaba i ja podem donar una resposta a l'usuari. En canvi, què passa si no hi ha cap habitació lliure? És a dir, què passaria si no es posa a *true* la variable `$trobat`? Doncs hem de tenir en compte aquesta situació a l'hora d'elaborar la condició de sortida del bucle. Si no ho tenim en compte el bucle mai acabaria i continuaria indefinidament.

Quina altra condició podem utilitzar per sortir del bucle si no hi ha cap habitació buida? És clar, haver recorregut tota la matriu. En aquest cas, quan haguem arribat

al cinquè pis. Si arribem a aquest pis, aquesta condició $\$pis < NUM_PLANTES$ esdevé falsa i el bucle s'atura.

Només ens faltarà comunicar a l'usuari si existeixen o no habitacions lliures.

```
1 echo ($trobat)? "Almenys hi ha una habitació lliure." : "No existeixen
   habitacions lliures.;"
```

Amb el codi anterior estem realitzant una estructura condicional alternativa al típic `if(...else....)`. S'anomena **operador ternari** i ens estalvia molt temps en l'escriptura del nostre codi, alhora que el simplifica.

```
1 expressio1 ? expressio2 : expressio3
```

S'avalua l'`expressio1`, i si el seu resultat és veritat, llavors s'avalua i retorna com a resultat l'`expressio2`. Si `expressio1` és fals, s'avalua i retorna `expressio3`. En aquest cas, és equivalent a escriure:

```
1 if (expressio1) {
2     expressio2;
3 } else {
4     expressio3;
5 }
```

Un exemple d'assignació a una variable:

```
1 //sense operador ternari
2 $faFred=false;
3 if ($faFred) {
4     $temps = "Fa molt de fred";
5 } else {
6     $temps = "No fa fred";
7 }
8 echo $temps;
9 //amb operador ternari
10 $faFred=false;
11 $temps = ($faFred)? "Fa molt de fred" : "No fa fred";
12 echo $temps;
```

Així, el codi equivalent a l'operador ternari utilitzat a la solució seria:

```
1 //operador ternari utilitzat a la solució
2 echo ($trobat)? "Almenys hi ha una habitació lliure." : "No existeixen
   habitacions lliures.;"
```

3

```
4 //estructura equivalent amb if...else
5 if($trobat){
6     echo "Almenys hi ha una habitació lliure.";
7 }
8 else{
9     echo "No existeixen habitacions lliures.";
10 }
```

2.3 Descrivint una persona a JSP, JSTL

En aquest apartat començarem a utilitzar el llenguatge de marques de Java: JSP/JSTL, que pertany a JEE7. Veurem quins tipus de dades existeixen a JSP/JSTL, com crear variables i quin és el seu àmbit. Aprendrem aquests conceptes en el procés de descriure una persona i intentar emmagatzemar les seves dades.

2.3.1 Sintaxi estàndard JSP

Als annexos del material didàctic trobareu el codi JSP/JSTL que utilitzarem en aquest apartat, per poder-lo descarregar. Recordeu que podeu utilitzar la funció d'importar del Netbeans per accedir a aquest contingut més fàcilment.

Comencem creant un projecte amb Netbeans, escollint l'opció *Maven / Web Application*. Crear un projecte amb Maven, com sabeu, és mot útil. És una eina capaç de generar totes les estructures de directoris per al projecte, descarregar automàticament les llibreries que s'utilitzin i, a més a més, està integrat a la majoria dels IDE.

Una vegada s'han descarregat les dependències del projecte, començarem afegint una pàgina JSP. Amb aquesta pàgina descriurem una persona.

A l'hora de crear la pàgina JSP ens demana la informació que podeu veure a la figura figura 2.3.

FIGURA 2.3. Creació d'una Java Servlet Page



De moment, seleccionem l'opció JSP File (Standard Syntax). Ens apareixerà un codi com aquest:

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     <title>JSP Page</title>
7   </head>
```

```

8   <body>
9     <h1>Hello World!</h1>
10    </body>
11 </html>

```

Sembla una pàgina web normal, però hi ha etiquetes addicionals. Per exemple, fixeu-vos-hi:

```

1  <%@page contentType="text/html" pageEncoding="UTF-8"%>

```

El codi JSP s'introdueix utilitzant les etiquetes: **<%— codi JSP —%>**. Dintre d'aquestes etiquetes podem utilitzar codi Java. Recordeu que aquest codi s'executa en el servidor abans d'enviar la pàgina al navegador. En aquest cas, **<%@page** és una **directiva** on es defineixen uns paràmetres que s'han de complir a tota la pàgina. En concret, es defineix el tipus de contingut de la pàgina JSP i la seva codificació.

També podem fer altres coses, com importar classes o bé indicar si volem activar la sessió de l'usuari. Exemple:

```

1  <%@page session="true" import="java.io.* , miPackage.miClase"%>

```

Per descriure les dades d'una persona necessitem declarar les variables necessàries per a la seva descripció. Fixeu-vos com es declaren les variables:

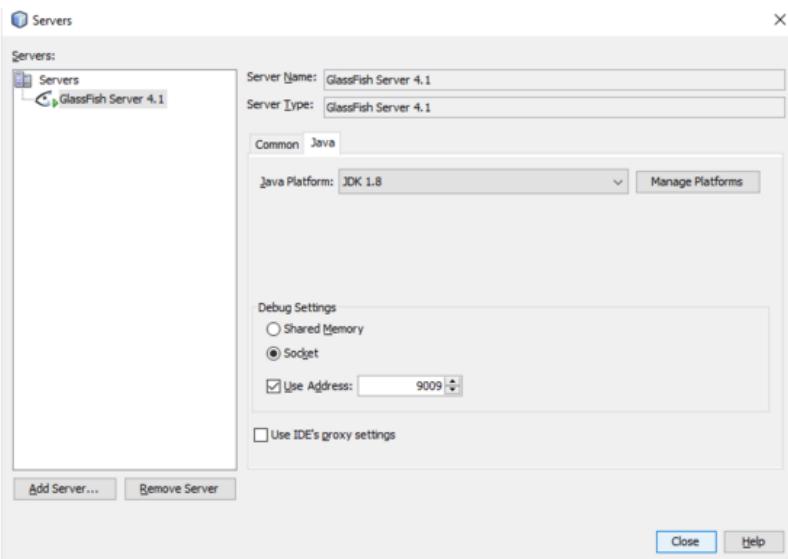
```

1  <%@page import="java.time.LocalDate"%>
2  <%@page contentType="text/html" pageEncoding="UTF-8"%>
3  <%!
4  String nomPersona ="Clara" ;
5  String cognoms ="Oswin";
6  int edat = 30;
7  LocalDate dataNaixement = LocalDate.of(1986,3,11);
8  String telf = "935555555";
9  String adrecaPostal = "Blackpool, England";
10 String email = "oswin@dr.who";
11 boolean treballa = false;
12 float alcada = 167.23f;
13 %>
14 <!DOCTYPE html>
15 <html>
16   <head>
17     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
18     <title>JSP Descrivint a una Persona</title>
19   </head>
20   <body>
21     <h1>Descrivint una persona</h1>
22     <h2> Les dades de la persona són:</h2>
23     <ul>
24       <li>Es diu: <%=nomPersona + " " + cognoms%> </li>
25       <li>Té <%=edat%> anys</li>
26       <li>Va néixer l'any: <%=dataNaixement.toString()%> </li>
27       <li>El seu telèfon és: <%=telf %> </li>
28       <li>Viu a <%=adrecaPostal %> </li>
29       <li>El seu e-mail és el <%=email %> </li>
30       <li>i actualment <%=(treballa)?"si":"no" %> treballa.</li>
31     </ul>
32   </body>
33 </html>

```

Per executar el codi anterior es fa servir la llibreria *java.time.LocalDate*, amb la qual és molt més senzill fer càcul amb dates. Per poder utilitzar-la s'ha de canviar la versió del JDK del servidor GlassFish del 1.7 al 1.8. Vegeu la figura figura 2.4.

FIGURA 2.4. Canvi de la versió del JDK al servidor Glassfish



L'execució del codi anterior la podeu veure a la figura figura 2.5.

FIGURA 2.5. Descrivint una persona amb JSP



Podeu executar el codi directament executant aquest fitxer (botó dret del ratolí damunt del fitxer i seleccionar *Run File*). El codi HTML resultant és:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
5          <title>JSP Descrivint a una Persona</title>
6      </head>
7      <body>
8          <h1>Descrivint una persona</h1>
9          <h2> Les dades de la persona són:</h2>
10         <ul>
11             <li>Es diu: Clara Oswin </li>
12             <li>Té 30 anys</li>
13             <li>Va néixer l'any: 1986-03-11 </li>
14             <li>El seu telèfon és el: 935555555 </li>
15             <li>Viu a Blackpool, England </li>
16             <li>El seu e-mail és oswin@dr.who </li>
17             <li>i actualment no treballa.</li>

```

```

18      </ul>
19    </body>
20 </html>
```

Per declarar les variables hem utilitzat el símbol **<%! –declaracions– %>**:

```

1 <%
2 String nomPersona ="Clara" ;
3 String cognoms ="Oswin";
4 int edat = 30;
5 LocalDate dataNaixement = LocalDate.of(1986,3,11);
6 String telf = "935555555";
7 String adrecaPostal = "Blackpool, England";
8 String email = "oswin@dr.who";
9 boolean treballa = false;
10 float alcada = 167.23f;
11 %>
```

Com que es necessita la llibreria *LocalDate* per declarar la data de naixement s'ha hagut d'importar, declarant-la al principi del document per poder-la utilitzar:

```
1 <%@page import="java.time.LocalDate"%>
```

Una vegada tenim les variables inicialitzades les utilitzem amb el símbol **<%= –expressió que retorna un valor –%>**.

```

1 ...
2 <li>Es diu: <%=nomPersona + " " + cognoms%> </li>
3 ...
4 <li>i actualment <%=(treballa)? "si": "no" %>    treballa.</li>
```

Aquest símbol **<%=** necessita d'una dada que mostrar. No permet executar més instruccions, només una, i aquesta ha de retornar un valor que es pugui mostrar en aquella posició en la pàgina JSP.

També podíem haver codificat la descripció de la persona de la següent manera:

```

1 <%@page import="java.time.LocalDate"%>
2 <%@page contentType="text/html" pageEncoding="UTF-8"%>
3 <%
4     String nomPersona = "Clara";
5     String cognoms = "Oswin";
6     int edat = 30;
7     LocalDate dataNaixement = LocalDate.of(1986, 3, 11);
8     String telf = "935555555";
9     String adrecaPostal = "Blackpool, England";
10    String email = "oswin@dr.who";
11    boolean treballa = false;
12    float alcada = 167.23f;
13  %>
14  <!DOCTYPE html>
15  <html>
16    <head>
17      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
18      <title>JSP Descrivint a una Persona</title>
19    </head>
20    <body>
21      <h1>Descrivint una persona</h1>
22      <h2> Les dades de la persona són:</h2>
23      <ul>
24        <%
25          String html = "<li>Es diu: " + nomPersona + " " + cognoms + " </li>
";
```

```

26     html += "<li>Té " + edat + " anys</li>";
27     html += "<li>Va néixer l'any:" + dataNaixement.toString() + "</li>";
28     html += "<li>El seu telèfon és el: " + telf + "</li>";
29     html += "<li>Viu a " + adrecaPostal + " </li>";
30     html += "<li>El seu e-mail és " + email + " </li>";
31     html += "<li> actualment " + ((treballa) ? "si" : "no") + "
32         treballa.</li>";
33     html += "<li>i té una alçada de " + alcada + " cm.</li>";
34     out.println(html);
35     %>
36   </body>
37 </html>
```

Veiem que amb el símbol `<% -codi java - %>` podem crear variables, podem executar instruccions i podem enviar el seu valor amb la instrucció `out.println(expr)` cap al navegador.

La pregunta que ens podem fer és: quina diferència existeix entre declarar les variables dintre dels símbols `<%! ... %>` o dintre de `<% ... %>`? Proveu el següent codi:

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%! int comptador=0; %>
3 <!DOCTYPE html>
4 <html>
5   <head>
6     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7     <title>Comptador</title>
8   </head>
9   <body>
10    <h1>Hola has executat aquesta pàgina <%= comptador++ %> vegades.</h1>
11  </body>
12 </html>
```

Si executeu la pàgina anterior veureu al navegador la següent informació:

1 Hola has executat aquesta pàgina 0 vegades.

Si, sense tancar la pàgina del navegador, premem F5 (la tornem a carregar), veurem:

1 Hola has executat aquesta pàgina 1 vegades.

La variable `comptador` no perd el seu valor, es manté. Aquest tipus de variables conserven el seu valor entre successives execucions. En canvi, si les declarem dintre de `<%..%>` no es manté el seu valor. Exemple:

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     <title>Comptador</title>
7   </head>
8   <body>
9     <%
10      int comptador = 0;
11      %>
12      <h1>Hola, has executat aquesta pàgina <%= comptador++ %> vegades.</h1>
13    </body>
```

```
14 </html>
```

Si executem el codi anterior i premem F5 moltes vegades, no canviarà el seu valor. Sempre mostrarà:

```
1 Hola, has executat aquesta pàgina 0 vegades.
```

Resum etiquetes JSP (Standard Syntax)

Una vegada hem arribat a aquest punt, recapitulem el tipus d'informació i sintaxi que podem utilitzar amb JSP (Standard Syntax):

- **Directives** (`<%@..%>`): indiquen informació general de la pàgina, com pot ser importació de classes, pàgina a invocar davant errors, si la pàgina forma part d'una sessió, incloure una altre pàgina o fitxer, etc.

Exemple:

```
1 <%@page session="true" import="java.util.ArrayList"%>
2 <%@include file="titulo.txt"%>
```

- **Declaracions** (`<%! .. %>`): serveixen per a funcions mètodes o variables. Aquestes variables conservaran el seu valor entre successives execucions. A més a més, les variables declarades dintre d'aquestes etiquetes esdevenen variables globals i poden ser utilitzades des de qualsevol lloc de la pàgina JSP.

```
1 <%-- Exemple --%>
2 <%!
3 private String ahora()
4 {
5     return ""+new java.util.Date();
6 }
7 %>
8 //Exemple2
9 <%! int contador=0; %>
```

- **Scriptlets** (`<%...%>`): codi Java embedut.

```
1 <%-- Exemple --%>
2 <table>
3     <% for (int i=0;i<10;i++) {
4         %>
5         <tr><td> <%=i%> </td></tr>
6     <% }
7         %>
8 </table>
```

- **Expressions** (`<%=...%>`): expressions Java que s'avaluen i s'envien a la sortida.

```
1 <%= new java.util.Date() %>
2 <%= Math.PI*2 %>
```

2.3.2 Sintaxi JSP Standard Tag Library (JSTL)

Veurem una altra manera d'afegir codi Java a una pàgina web i utilitzarem les llibreries d'etiquetes JSTL. Aquesta manera de definir una pàgina web està basada en etiquetes XML.

La llibreria estàndard d'etiquetes **JSTL** és una col·lecció d'etiquetes JSP que encapsula les funcionalitats principals de moltes aplicacions JSP.

Començarem creant una nova pàgina web JSP i afegirem el conjunt d'etiquetes principals JSTL. Per accedir al nou conjunt d'etiquetes posarem el prefix *c*.

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

A més, com que utilitzarem dates necessitem el conjunt d'etiquetes *i18n* (internacionalització), que ens permetrà utilitzar-les d'una manera més còmoda.

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

Una vegada hem afegit les llibreries de les etiquetes XML que utilitzarem comencem a definir les variables i a mostrar el seu valor per pantalla. El codi resultant és el següent:

```
1 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
3 <%@page contentType="text/html" pageEncoding="UTF-8"%>
4 <!DOCTYPE html>
5 <html>
6   <head>
7     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8     <title>JSTL Descrivint a una Persona</title>
9   </head>
10  <body>
11    <h1>Descrivint una persona</h1>
12    <h2> Les dades de la persona són:</h2>
13    <c:set var="nom" value="Clara" scope="page" />
14    <c:set var="cognoms" value="Oswin" scope="page" />
15    <c:set var="edat" value="30" scope="page" />
16    <fmt:parseDate var="datanaixement" pattern="dd-MM-yyyy" value=
17      "11-03-1986"/>
18    <c:set var="telef" value="935555555" scope="page" />
19    <c:set var="adreca" value="Blackpool, England" scope="page" />
20    <c:set var="email" value="oswin@dr.who" scope="page" />
21    <c:set var="treballa" value="false" scope="page" />
22    <c:set var="alcada" value="167.23f" scope="page" />
23    <ul>
24      <li>Es diu: <c:out value="${nom} ${cognoms}" /></li>
25      <li>Té <c:out value="${edat}" /> anys</li>
26      <li>Va néixer el <fmt:formatDate value="${datanaixement}"
27        dateStyle="full"/>.</li>
28      <li>El seu telèfon és el: <c:out value="${telef}" /> </li>
29      <li>Viu a <c:out value="${adreca}" /> </li>
30      <li>El seu e-mail és <c:out value="${email}" /> </li>
31      <li> actualment <c:out value="${treballa? 'si' : 'no'}" />
32        treballa.</li>
33      <li>i té una alçada de <c:out value="${alcada + 1}" /> cm.</li>
34    </ul>
35  </body>
```

33 </html>

Com veieu, per declarar una variable utilitzem l'etiqueta **<c:set>**. Recordeu que hem utilitzat la lletra **c** com a prefix per poder utilitzar les etiquetes del nucli *jstl*, i per això les etiquetes comencen amb **<c:>**. Una vegada escollida l'etiqueta per declarar variables, a continuació hem de donar-li un nom, un valor i un àmbit. Per donar-li el nom utilitzem l'atribut de l'etiqueta *var*, per donar-li un valor utilitzem l'atribut *value*, i per donar-li un àmbit a la variable utilitzem l'atribut *scope*. Exemple:

1 <c:set var="edat" value="30" scope="page" />

Com podeu veure en l'exemple anterior, no cal definir el tipus de variable. Automàticament es converteix en el tipus que necessitem per fer l'operació que necessitem. Vegeu com s'utilitza la variable *alcada*.

Per mostrar la variable per pantalla utilitzem l'etiqueta **<c:out>**. Aquesta etiqueta només té l'atribut *value*, amb el qual podem mostrar per pantalla el valor d'una variable o d'una expressió. Per poder utilitzar una variable de les definides prèviament s'ha de fer servir la sintaxi **\${expr}**, on *expr* pot ser el nom de la variable a mostrar. Aquest nom correspon a l'atribut *var* de l'etiqueta **<c:set>**. Exemple:

1 i té una alçada de <c:out value="\${alcada + 1}" /> cm.

A l'exemple anterior agafem el valor d'una variable i li sumem 1. Aquest valor es veurà per pantalla en aquesta posició. Com veieu, no cal definir els tipus; automàticament s'ha convertit en un *float*, ha sumat un 1 i després l'ha convertit en un *string*.

Per utilitzar les variables creades s'utilitza la notació **\${expr}**, que és estàndard i molt potent. És un llenguatge en si mateix que s'anomena *EL* (Expression Language).

EL (Expression Language) permet la resolució dinàmica dels objectes i els mètodes de Java en pàgines JSP i Facelets. Les expressions *EL* tenen la forma de **\${foo}** i **#{bar}**.

JavaServer Faces

JavaServer Faces (**JSF**) és una tecnologia per a aplicacions Java basades en web que simplifica el desenvolupament d'aplicacions Java EE. El principal avantatge sobre JSP és que permet una clara separació entre el comportament de l'aplicació i la seva presentació.

A més a més, per poder programar amb aquest llenguatge necessitem saber quines són les etiquetes que ens proporciona la llibreria principal JSTL (vegeu la taula 2.2).

TAULA 2.2. Etiquetes de la llibreria JSTL

Etiqueta	Significat
<i>set</i>	S'utilitza per donar un valor a una propietat o a una variable en qualsevol dels seus àmbits
<i>remove</i>	S'utilitza per eliminar una variable
<i>choose</i>	És una etiqueta condicional utilitzada per a la creació d'un condicional del tipus <i>if...else</i>

TAULA 2.2 (continuació)

Eiqueta	Significat
<i>when</i>	Aquesta etiqueta s'utilitza dintre de l'etiqueta <i>choose</i> . Serveix per avaluar una expressió a <i>true</i> o <i>false</i>
<i>otherwise</i>	Aquesta etiqueta s'utilitza dintre de l'etiqueta <i>choose</i> . El contingut d'aquesta etiqueta s'executa si l'expressió avaluada per l'etiqueta <i>when</i> és <i>false</i>
<i>forEach</i>	Aquesta etiqueta serveix per fer un bucle
<i>forTokens</i>	Aquesta etiqueta serveix per iterar sobre un <i>string</i> . A cada volta dóna la porció de l' <i>string</i> que hi ha entre uns delimitadors
<i>import</i>	Aquesta etiqueta serveix per importar un fitxer. Per exemple, podem importar el contingut d'una pàgina o un fitxer dintre d'una variable per després tractar la informació
<i>param</i>	S'utilitza per afegir un paràmetre a la URL
<i>redirect</i>	S'utilitza per redirigir a l'usuari a una altra URL
<i>url</i>	S'utilitza per crear una URL a una pàgina
<i>catch</i>	Serveix per agafar els errors que es produeixen a la pàgina
<i>out</i>	Aquesta etiqueta és l'equivalent a les expressions <%=..%> de la llibreria estàndard de JSP

Anirem utilitzant les etiquetes anteriors segons les requerim per solucionar els diversos problemes que se'n presentin a l'hora de codificar els exemples amb JSTL.

Només queda per explicar l'atribut *scope* de l'etiqueta *set*. Aquest atribut no deixa de ser l'àmbit de la variable declarada. A JSP existeixen diferents àmbits (taula 2.3).

TAULA 2.3. Tipus d'àmbits a JSP i JSTL

Tipus d'àmbit	Significat
<i>page</i>	Els objectes declarats en aquest àmbit només són accessibles en la pàgina declarada. La dada d'aquest objecte només és vàlida durant el processament de l'actual resposta.
<i>request</i>	Els objectes d'àmbit de <i>request</i> són accessibles en les pàgines processades en la mateixa petició en la qual es va crear.
<i>session</i>	Els objectes declarats en aquest àmbit són accessibles en les pàgines processades de sol·licituds que es troben en la mateixa sessió que en la que van crear.
<i>application</i>	Els objectes declarats en aquest àmbit són accessibles per qualsevol pàgina JSP que formi part de la mateixa aplicació.

Millor veiem amb un exemple com s'utilitzen:

```

1 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2
3 <html>
4   <head>
5     <title>Àmbits de les variables a JSOLT</title>
```

```

6   </head>
7   <body>
8       <c:set var="test" value="Valor d'una variable d'àmbit de pàgina" scope=
9           "page" />
10      <c:set var="test" value="Valor d'una variable d'àmbit de Request"
11          scope="request" />
12      <c:set var="test" value="Valor d'una variable d'àmbit de sessió"
13          scope="session" />
14
15      <c:set var="test" value="Valor d'una variable d'àmbit d'aplicació"
16          scope="application" />
17
18      <h1>Àmbits de les variables a JSLT:</h1>
19      <table border="1">
20          <tr>
21              <td>
22                  <b>Àmbit per defecte: </b>
23              </td>
24              <td>
25                  <c:out value="${test}" />
26              </td>
27          </tr>
28          <tr>
29              <td>
30                  <b>Àmbit de pàgina</b>
31              </td>
32              <td>
33                  <c:out value="${pageScope.test}" />
34              </td>
35          </tr>
36          <tr>
37              <td>
38                  <b>Àmbit de Request: </b>
39              </td>
40              <td>
41                  <c:out value="${requestScope.test}" />
42              </td>
43          </tr>
44          <tr>
45              <td>
46                  <b>Àmbit de sessió</b>
47              </td>
48              <td>
49                  <c:out value="${sessionScope.test}" />
50              </td>
51          </tr>
52          <tr>
53              <td>
54                  <b>Àmbit d'aplicació</b>
55              </td>
56              <td>
57                  <c:out value="${applicationScope.test}" />
58              </td>
59          </tr>
60      </table>
61  </body>
62 </html>

```

Observeu que el nom de la variable utilitzada és el mateix (*test*). Aquesta variable té diferents valors depenen de l'àmbit on està. En altres paraules, existeixen diferents variables anomenades *test* en diferents àmbits del programa.

En executar el codi anterior obtenim la pàgina que podeu veure a la figura figura 2.6

FIGURA 2.6. Àmbits de les variables en JSP

The screenshot shows a browser window with the URL `localhost:8080/m7-a2-jsp/ambitsJSLT.jsp`. The page title is "Àmbits de les variables a JSTL/JSP:". Below the title is a table with five rows, each containing a scope name and its corresponding value:

Àmbit per defecte:	Valor d'una variable d'àmbit de pàgina
Àmbit de pàgina	Valor d'una variable d'àmbit de pàgina
Àmbit de Request	Valor d'una variable d'àmbit de Request
Àmbit de sessió	Valor d'una variable d'àmbit de sessió
Àmbit d'aplicació	Valor d'una variable d'àmbit d'aplicació

2.3.3 Calcular l'edat d'una persona

Continuant amb l'exemple de descripció d'una persona, volem afegir la funcionalitat de calcular l'edat si sabem l'any que va néixer. Volem fer el càlcul de dues maneres: amb la sintaxi estàndard i amb la sintaxi JSTL.

Començarem amb la sintaxi estàndard. Afegirem una funció que rebi per paràmetre la data de naixement i retorni l'edat de la persona fins al dia actual.

```

1 <%@page import="java.time.temporal.ChronoUnit"%>
2 <%@page import="java.time.LocalDate"%>
3 <%@page contentType="text/html" pageEncoding="UTF-8"%>
4 <%
5 private long calculaEdat(LocalDate dataNaix){
6     LocalDate now = LocalDate.now();
7     return ChronoUnit.YEARS.between(dataNaix, now);
8 }
9 %>
10 <!DOCTYPE html>
11 <html>
12     <head>
13         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
14         <title>JSP Descrivint a una Persona</title>
15     </head>
16     <body>
17         <h1>Descrivint una persona</h1>
18         <h2> Les dades de la persona són:</h2>
19         <ul>
20             <%
21                 String nomPersona = "Clara";
22                 String cognoms = "Oswin";
23                 LocalDate dataNaixement = LocalDate.of(1986, 3, 11);
24             %>
25             <li>Es diu: <%=nomPersona + " " + cognoms%> </li>
26             <li>Té <%=calculaEdat(dataNaixement)%> anys</li>
27             <li>Va néixer l'any: <%=dataNaixement.toString()%> </li>
28         </ul>
29     </body>
30 </html>
```

La declaració de la funció es fa dintre de la zona de declaracions. Una vegada s'ha declarat la funció es pot utilitzar en qualsevol lloc del codi JSP. En aquest cas, la utilitzem directament a una expressió per tal de retornar l'edat de la persona i mostrar-la en el lloc adient de la pàgina JSP.

La variable `dataNaixement` és una variable local que només es pot fer servir en el mateix *scriptlet* i en les expressions d'assignació (`<%=`). Per poder utilitzar aquest valor dintre d'una funció s'ha d'enviar mitjançant un paràmetre, tal com s'ha fet a l'exemple.

També podríem haver utilitzat variables globals. No queda tan bé, però es pot implementar. Exemple:

```

1 <%@page import="java.time.temporal.ChronoUnit"%>
2 <%@page import="java.time.LocalDate"%>
3 <%@page contentType="text/html" pageEncoding="UTF-8"%>
4 <%
5 public long calculaEdat(){
6     LocalDate now = LocalDate.now();
7     return ChronoUnit.YEARS.between(dataNaixement, now);
8 }
9 LocalDate dataNaixement = LocalDate.of(1986, 3, 11);
10 %>
11 <!DOCTYPE html>
12 <html>
13     <head>
14         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
15         <title>JSP Descrivint a una Persona</title>
16     </head>
17     <body>
18         <h1>Descrivint una persona</h1>
19         <h2> Les dades de la persona són:</h2>
20         <ul>
21             <%
22                 String nomPersona = "Clara";
23                 String cognoms = "Oswin";
24             %>
25             <li>Es diu: <%=nomPersona + " " + cognoms%> </li>
26             <li>Té <%=calculaEdat()%> anys</li>
27             <li>Va néixer l'any: <%=dataNaixement.toString()%> </li>
28         </ul>
29     </body>
30 </html>
```

La variable global *dataNaixement* s'ha declarat dintre de la zona dedicada a la declaracions de variables i funcions. Aquesta variable, en ser global, es pot utilitzar perfectament dintre de la funció (sense haver d'enviar-la per paràmetre) i en les expressions d'assignacions (<%=).

La segona manera de calcular l'edat d'una persona és utilitzant etiquetes JSTL.

```

1 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
3 <%@page contentType="text/html" pageEncoding="UTF-8"%>
4 <!DOCTYPE html>
5 <html>
6     <head>
7         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8         <title>JSTL Descrivint a una Persona</title>
9     </head>
10    <body>
11        <h1>Descrivint una persona</h1>
12        <h2> Les dades de la persona són:</h2>
13        <c:set var="nom" value="Clara" scope="page" />
14        <c:set var="cognoms" value="Oswin" scope="page" />
15        <fmt:parseDate var="datanaixement" pattern="dd-MM-yyyy" value="
16            11-03-1986"/>
17        <c:set var="avui" value="<%=new java.util.Date()%>" />
18        <ul>
19            <li>Es diu: <c:out value="${nom} ${cognoms}" /></li>
20            <li>Té <c:out value="${avui.year - datanaixement.year}" /> anys<
21                li>
22        </ul>
23    </body>
</html>
```

Observem que en aquesta solució només es té en compte, en el càlcul de l'edat de la persona, els anys de diferència que hi ha entre les dues dates. Primer creem la data de la persona amb l'etiqueta *fmt*.

```
1 <fmt:parseDate var="datanaixement" pattern="dd-MM-yyyy" value="11-03-1986"/>
```

Aquesta etiqueta crea una data a la variable *datanaixement*. Aquesta data té l'any de naixement de la persona en el format *dia-mes-any*. També podem crear dates utilitzant etiquetes JSTL amb etiquetes JSP estàndard. Fixeu-vos com hem creat la data actual:

```
1 <c:set var="avui" value="<%new java.util.Date()%>" />
```

Amb el codi anterior hem creat una variable anomenada *avui* que conté la data actual. Ara només s'ha de calcular la diferència entre aquestes dues dates per tal d'obtenir l'edat de la persona.

```
1 <c:out value="${avui.year - datanaixement.year}" />
```

El codi anterior no fa el càlcul exacte de l'edat, però per començar ja ens va prou bé. Fa la resta exacta entre l'any de naixement i l'any actual sense tenir en compte el mes i el dia. L'interessant d'aquesta notació és que podem accedir als *getters* de la classe `java.util.Date` directament, sense haver de posar `datanaixement.getYear()`. Aquesta notació ens permet accedir a la propietat d'una classe si aquesta té la funció *get* associada.

2.4 Gestionant un hotel amb JSP

En aquest apartat veurem el tipus de dades *array* i les estructures de control necessàries per tractar amb aquest tipus de dades. Aprendrem aquests conceptes gestionant, d'una manera molt senzilla, alguns aspectes de la gestió d'un hotel.

2.4.1 Gestió de les taules d'un restaurant

Igual que vam fer amb PHP, volem utilitzar el llenguatge JSP per guardar el nombre de comensals que hi ha en cada taula i mostrar la informació.

Penseu com determinar quines taules estan buides. Volem crear una funció que donat un *array* amb el nombre de comensals de cada taula ens retorna en un altre *array* el nombre de cada taula que estigui buida.

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%
3     private int rand(int min, int max) {
4         return (int) (Math.random() * (max+1 - min) + min);
5     }
```

```

6  %>
7  <!DOCTYPE html>
8  <html>
9    <head>
10       <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
11       <title>Gestió de taules d'un restaurant</title>
12    </head>
13    <body>
14      <%
15          //inicialització de les taules amb els comensals
16          int[] taules = new int[10];
17          for (int i = 0; i < 10; i++) {
18              taules[i] = rand(0, 5);
19          }
20      %>
21      <h1>Gestió de taules d'un restaurant</h1>
22      <ul>
23          <%
24              //llistat de les taules
25              for(int i = 0; i < 10; i++)
26              {
27                  switch(taules[i]){
28                      case 0:
29                          %>
30                          <li>La taula <%=i%> està buida</li>
31                          <%
32                          break;
33                      case 5:
34                          %>
35                          <li>La taula <%=i%> està plena.</li>
36                          <%
37                          break;
38                      default:
39                          %>
40                          <li>La taula <%=i%> té <%=taules[i]%> comensals</li
41                          >
42                  }
43              }
44          %>
45      </ul>
46  </body>
47 </html>

```

Una possible resposta a l'execució del codi anterior és el següent:

```

1 Gestió de taules d'un restaurant
2
3 La taula 0 està plena.
4 La taula 1 té 1 comensals
5 La taula 2 està plena.
6 La taula 3 té 4 comensals
7 La taula 4 té 1 comensals
8 La taula 5 està buida
9 La taula 6 té 1 comensals
10 La taula 7 està buida
11 La taula 8 està buida
12 La taula 9 té 3 comensals

```

Observeu el codi anterior. Veieu que s'ha utilitzat l'estruatura **for** per recórrer l'*array*. Aquesta estructura funciona igual que a Java i a PHP. El que pot sobtar és la manera d'enviar la informació al navegador. Observeu amb deteniment l'estruatura del bucle **for** i com utilitzem les etiquetes **<% ... %>**.

```

1 <%
2 for (int i = 0; i < 10; i++) {

```

```

3   switch (taules[i]) {
4     case 0:
5       %>
6         <li>La taula <%=i%> està buida</li>
7       <%
8         break;
9       case 5:
10      %>
11        <li>La taula <%=i%> està plena.</li>
12      <%
13        break;
14      default:
15      %>
16        <li>La taula <%=i%> té <%=taules[i]%> comensals</li>
17      <%
18    }
19  }
20 %>

```

En comptes d'utilitzar la funció *out.println(expr)* per enviar la informació al navegador hem tancat l'etiqueta JSP i hem posat el codi HTML directament a la pàgina. Perquè el codi es pugui enviar a la pàgina HTML ha de complir les condicions posades anteriorment dintre de les etiquetes JSP. Així, podem anar alternant entre codi HTML i codi JSP.

Una alternativa del codi anterior podria ser:

```

1 <%
2 for (int i = 0; i < 10; i++) {
3   switch (taules[i]) {
4     case 0:
5       out.println("<li>La taula " + i + " està buida</li>");
6
7       break;
8     case 5:
9       out.println("<li>La taula " + i + " està plena.</li>");
10
11      break;
12    default:
13      out.println("<li>La taula " + i + " té " + taules[i] +
14        " comensals.</li>");
15    }
16  }
%>

```

Solucionem el repte anterior. Volem crear una funció que donat un *array* amb el nombre de comensals de cada taula ens retorni en un altre *array* el nombre de cada taula que estigui buida. Mostrarem per pantalla totes les taules buides.

```

1 <%@page import="java.util.ArrayList"%>
2 <%@page contentType="text/html" pageEncoding="UTF-8"%>
3 <%
4   private int rand(int min, int max) {
5     return (int) (Math.random() * (max + 1 - min) + min);
6   }
7   private ArrayList cercarBuides(int[] taules){
8     ArrayList buides = new ArrayList();
9     for (int i = 0; i < taules.length; i++) {
10       if(taules[i] == 0){
11         buides.add(i);
12       }
13     }
14   }
15 }

```

```

16  %>
17  <!DOCTYPE html>
18  <html>
19      <head>
20          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
21          <title>Gestió de taules d'un restaurant</title>
22      </head>
23      <body>
24          <%
25              //inicialització de les taules amb els comensals
26              int[] taules = new int[10];
27              for (int i = 0; i < 10; i++) {
28                  taules[i] = rand(0, 5);
29              }
30              //cerquem les taules buides
31              ArrayList taulesBuides = cercarBuides(taules);
32          %>
33          <h1>Les següents taules estan buides:</h1>
34          <ul>
35              <%
36                  //llistat de les taules buides
37                  if(taulesBuides.isEmpty()){
38                      out.println("No hi han taules buides.");
39                  }
40                  for (Object taula: taulesBuides){
41                      out.println("<li>La taula " + taula + " està buida.</li>");
42                  }
43              %>
44          </ul>
45      </body>
46  </html>

```

Una possible resposta a l'execució del codi anterior seria:

```

1 Les següents taules estan buides:
2
3 La taula 1 està buida.
4 La taula 6 està buida.
5 La taula 9 està buida.

```

Hem emprat la classe `ArrayList` per emmagatzemar les taules buides. Com que, a priori, no sabem quantes taules buides tindrem, no podem utilitzar un *array* de mida fixa. Java ens proporciona la classe `ArrayList` per a aquests casos. La funció `cercarBuides` recorre l'*array* `taules` per cercar el número de taula que té 0 comensals. Quan trobem una taula que no té cap comensal l'afegeim a l'*array* de taules buides.

```

1 private ArrayList cercarBuides(int[] taules){
2     ArrayList buides = new ArrayList();
3     for (int i = 0; i < taules.length; i++) {
4         if(taules[i] == 0){
5             buides.add(i);
6         }
7     }
8     return buides;
9 }

```

Quan hem obtingut l'`ArrayList` amb el llistat de taules buides hem de mostrar-lo per pantalla. Aquesta vegada iterem l'`ArrayList` amb un bucle del tipus *foreach*. A Java, aquests bucles es codifiquen de la següent manera:

```

1 for (Object taula: taulesBuides){
2     out.println("<li>La taula " + taula + " està buida.</li>");

```

3 } _____

Si ens hi fixem, és un bucle **for** però amb una altra sintaxi. Aquesta vegada, dintre dels parèntesis hem d'indicar el tipus d'element que tindrem a cada iteració, el nom de la variable que contindrà l'element i la col·lecció d'elements (podria ser un *array*, però en el nostre cas és un *ArrayList*). En general:

```
1 for(Tipus element: col·lecció){
2     //fer alguna cosa amb l'element
3 }
```

Ja hem vist com utilitzar *arrays* d'una dimensió amb JSP, i ara veurem els mateixos exemples però amb la sintaxi JSTL. Veureu que no tot es pot fer amb JSTL, però farem tot el possible per utilitzar-ho al màxim.

Començarem amb la implementació del programa que llista les taules d'un restaurant.

```
1 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2 <%@page contentType="text/html" pageEncoding="UTF-8"%>
3 <%
4     private int rand(int min, int max) {
5         return (int) (Math.random() * (max + 1 - min) + min);
6     }
7 %>
8 <!DOCTYPE html>
9 <html>
10    <head>
11        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12        <title>Gestió de taules d'un restaurant</title>
13    </head>
14    <body>
15        <%
16            //inicialització de les taules amb els comensals
17            int[] taules = new int[10];
18            for (int i = 0; i < 10; i++) {
19                taules[i] = rand(0, 5);
20            }
21            request.setAttribute("taules", taules);
22        %>
23        <h1>Gestió de taules d'un restaurant</h1>
24        <ul>
25            <c:forEach var="num" varStatus="loop" items="${taules}">
26                <c:choose>
27                    <c:when test="${num eq 0}">
28                        <c:out value="${'<li>La taula'} ${loop.index} ${'està
29                                buida</li>'}" escapeXml="false"></c:out>
30                    <c:when test="${num eq 5}">
31                        <c:out value="${'<li>La taula'} ${loop.index} ${' està
32                                plena</li>'}" escapeXml="false"></c:out>
33                    <c:otherwise>
34                        <c:out value="${'<li>La taula'} ${loop.index} ${' té '}
35                                ${num} ${'comensals</li>'}" escapeXml="false"></c
36                                :out>
37                    </c:otherwise>
38                </c:choose>
39            </c:forEach>
40        </ul>
41        </body>
42    </html>
```

Si observem detingudament el codi anterior ens adonarem que just després de crear l'*array taules* i omplir-lo amb els comensals s'ha enregistrat en l'objecte ***request***.

```

1 //inicialització de les taules amb els comensals
2 int[] taules = new int[10];
3 for (int i = 0; i < 10; i++) {
4     taules[i] = rand(0, 5);
5 }
6 request.setAttribute("taules", taules);

```

Els objectes que es creen amb la sintaxi estàndard JSP no es poden utilitzar, per defecte, amb les etiquetes JSTL. Aquests objectes se'ls ha de fer accessibles des de JSTL. Una manera de fer-lo accessible és afegint-lo com a atribut del objecte *request* de la pàgina. Una vegada hem fet accessible l'objecte el podem assignar a una variable de JSTL o utilitzar-lo com a col·lecció amb la qual poder iterar.

```

1 <c:forEach var="num" varStatus="loop" items="${taules}">
2 ...
3 </c:forEach>

```

És exactament el que hem fet amb l'*array taules*. L'hem assignat a l'atribut *items* de l'etiqueta *foreach*. Aquest atribut ha de ser una col·lecció amb la qual poder extreure tota la informació element a element. Cada element de l'*array taules* s'emmagatzema dintre de la variable *num*. A més a més, podem accedir a l'índex de l'*array* de la iteració actual. El nom de l'índex de l'*array* s'ha d'especificar en l'atribut *varStatus* de l'etiqueta *foreach*.

Una vegada ja tenim l'estructura del bucle hem d'utilitzar una estructura de *switch*. A JSTL aquesta estructura la podem crear utilitzant la combinació de les etiquetes *choose*, *when* i *otherwise*.

```

1 //estructura equivalent a switch() però amb etiquetes JSTL
2 <c:choose>
3     <c:when test="${num eq 0}">
4         ...
5     </c:when>
6     <c:when test="${num eq 5}">
7         ...
8     </c:when>
9     <c:otherwise>
10        ...
11    </c:otherwise>
12 </c:choose>

```

Podem utilitzar tantes etiquetes *when* com ens faci falta per especificar els *case* de l'estructura *switch*. L'únic atribut que admet aquesta etiqueta és *test*. Aquest atribut admet una EL (*Expression Language*) que en ser avaluada retorna *true* o *false*. Si s'avalua com a *true* s'executaràn les instruccions que hi ha dintre de la mateixa etiqueta *when*. En el cas que s'avalui a *false* anirà a avaluar l'atribut *test* de la següent etiqueta *when* fins que, finalment, s'executa l'etiqueta *otherwise*.

Per acabar l'exemple només ens queda imprimir per pantalla la informació que ens demana l'enunciat de l'exercici. Volem fer un llistat amb les taules i el nombre de comensals.

Per fer aquesta part només hem d'utilitzar les etiquetes *<c:out>* per enviar la informació al navegador.

```

1 <c:choose>
2   <c:when test="${num eq 0}">
3     <c:out value="${'<li>La taula' } ${loop.index} ${'està buida</li>'}"
4       escapeXml="false"></c:out>
5   </c:when>
6   <c:when test="${num eq 5}">
7     <c:out value="${'<li>La taula' } ${loop.index} ${' està plena</li>'}"
8       escapeXml="false"></c:out>
9   </c:when>
10  <c:otherwise>
11    <c:out value="${'<li>La taula' } ${loop.index} ${' té '} ${num} ${'
comensals</li>'}" escapeXml="false"></c:out>
12  </c:otherwise>
13 </c:choose>

```

Per imprimir correctament la informació veieu que s'ha utilitzat una concatenació d'expressions *EL*. Per imprimir un *string* s'han d'utilitzar les cometes simples. Dintre de l'*string* podem afegir etiquetes HTML perquè les interpreti el navegador sempre que l'atribut **escapeXML** estigui desactivat. Si no el desactivem, el navegador no interpretarà les etiquetes enviades amb l'etiqueta *<c:out>*.

Dintre de l'expressió *EL* podem utilitzar la variable de l'etiqueta *foreach* i l'índex de la col·lecció per mostrar la informació a l'usuari. L'execució del codi anterior ens dóna aquesta sortida:

```

1 Gestió de taules d'un restaurant
2
3 La taula 0 està plena
4 La taula 1 té 4 comensals
5 La taula 2 té 4 comensals
6 La taula 3 està buida
7 La taula 4 està buida
8 La taula 5 té 3 comensals
9 La taula 6 està plena
10 La taula 7 té 2 comensals
11 La taula 8 té 2 comensals
12 La taula 9 té 3 comensals

```

Intenteu implementar amb etiquetes JSTL, dintre del possible, el següent exercici: creareu una funció que donat un *array* amb el nombre de comensals de cada taula ens retorni un altre *array* amb el nombre de cada taula que estigui buida. Mostrareu per pantalla totes les taules buides.

2.4.2 Gestió de les habitacions d'un hotel

En aquest apartat es veurà el tractament d'informació amb *arrays* de més d'una dimensió. Aprendreu com crear *arrays* multidimensionals, com modificar els valors i com cercar-los.

Es tractarà de programar la gestió d'habitacions d'un hotel. Imagineu un hotel amb 5 plantes i 10 habitacions en cadascuna de les plantes. Es vol guardar el nombre de clients que hi ha en cada habitació, i com màxim n'hi pot haver 4.

Començarem creant un *array* de dues dimensions (matriu bidimensional) on a cada posició guardarem el nombre de clients que hi ha. Amb el primer índex de

l'*array* accedirem al pis, i amb el segon índex accedim a la habitació d'aquell pis. Per exemple, si accedim a la posició de l'*array* habitacions[0][4] estem accedint al pis 0 (planta baixa) habitació 4.

Doncs bé, igual que abans amb el restaurant, inicialitzarem l'*array* de manera aleatòria amb el nombre de clients per habitació. Una vegada tinguem la matriu preparada mostrarem per pantalla la informació.

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%
3     private static final int NUM_PLANTES = 5;
4     private static final int NUM_HAB = 10;
5     private static final int MAX_CLIENTS = 5;
6     private int rand(int min, int max) {
7         return (int) (Math.random() * (max + 1 - min) + min);
8     }
9
10    %>
11    <!DOCTYPE html>
12    <html>
13        <head>
14            <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
15            <title>Gestió de les habitacions d'un hotel</title>
16        </head>
17        <body>
18            <H1>Gestió de les habitacions d'un hotel</H1>
19            <%
20                //inicialització de les taules amb els comensals
21                int[][] hotel = new int[NUM_PLANTES][NUM_HAB];
22                for (int pis = 0; pis < NUM_PLANTES; pis++) {
23                    for (int hab = 0; hab < NUM_HAB; hab++) {
24                        hotel[pis][hab] = rand(0, MAX_CLIENTS);
25                    }
26                }
27            %>
28            <ul>
29            <%
30                //Llistat amb el nombre de clients per habitacions
31                for (int pis = 0; pis < NUM_PLANTES; pis++) {
32                    for (int hab = 0; hab < NUM_HAB; hab++) {
33                        out.println("<li>A l'habitació " + hab + " de la planta " +
34                                pis + " hi ha " + hotel[pis][hab] + " clients</li>");
35                    }
36                }
37            %>
38            </ul>
39        </body>
50    </html>
```

Una possible execució del codi anterior és la següent:

```

1 Gestió de les habitacions d'un hotel
2
3 A l'habitació 0 de la planta 0 hi ha 0 clients
4 A l'habitació 1 de la planta 0 hi ha 3 clients
5 A l'habitació 2 de la planta 0 hi ha 5 clients
6 A l'habitació 3 de la planta 0 hi ha 5 clients
7 A l'habitació 4 de la planta 0 hi ha 2 clients
8 A l'habitació 5 de la planta 0 hi ha 5 clients
9 A l'habitació 6 de la planta 0 hi ha 3 clients
10 A l'habitació 7 de la planta 0 hi ha 4 clients
11 A l'habitació 8 de la planta 0 hi ha 1 clients
12 A l'habitació 9 de la planta 0 hi ha 4 clients
13
14 ...
15
16 A l'habitació 0 de la planta 4 hi ha 2 clients
```

17 A l'habitatçó 1 de la planta 4 hi ha 3 clients
18 A l'habitatçó 2 de la planta 4 hi ha 4 clients
19 A l'habitatçó 3 de la planta 4 hi ha 0 clients
20 A l'habitatçó 4 de la planta 4 hi ha 2 clients
21 A l'habitatçó 5 de la planta 4 hi ha 1 clients
22 A l'habitatçó 6 de la planta 4 hi ha 5 clients
23 A l'habitatçó 7 de la planta 4 hi ha 2 clients
24 A l'habitatçó 8 de la planta 4 hi ha 5 clients
25 A l'habitatçó 9 de la planta 4 hi ha 2 clients

Fixem-nos en la declaració de les constants. A JSP s'han de declarar a l'espai de declaracions, és a dir, dintre de les etiquetes `<%! %>`. Les constants es declaren utilitzant les paraules reservades ***static*** i ***final***, tot i que només amb ***final***, en el nostre cas, ja ens serviria.

```
1 <%!
2     private static final int NUM_PLANTES = 5;
3     private static final int NUM_HAB = 10;
4     private static final int MAX_CLIENTS = 5;
5     private int rand(int min, int max) {
6         return (int) (Math.random() * (max + 1 - min) + min);
7     }
8 %>
```

La diferència entre posar *static* i no posar-ho és que les variables estàtiques (*static*) pertanyen a la classe, i en les no estàtiques hi ha una per a cada objecte de la classe. La paraula reservada *final* significa que el valor de la variable no canviará, és a dir, serà constant.

A l'hora de recórrer i de crear la matriu veiem que no hi ha sorpreses. Per crear la matriu utilitzem la instrucció:

```
1 int[][] hotel = new int[NUM_PLANTES][NUM_HAB];
```

Per crear la matriu hem de dir la mida que tindran els dos índexs. I per recórrer la matriu utilitzem dos bucles de tipus ***for***.

```
1 //Llistat amb el nombre de clients per habitacions
2 for (int pis = 0; pis < NUM_PLANTES; pis++) {
3     for (int hab = 0; hab < NUM_HAB; hab++) {
4         out.println("<li>A l'habitació " + hab + " de la planta " + pis + " hi
5             ha " + hotel[pis][hab] + " clients</li>");
6     }
7 }
```

Per acabar, volem un llistat amb l'ocupació de les habitacions de l'hotel i una funció que ens indiqui si hi ha habitacions lliures. Penseu com fer la funció mentre traiem el llistat amb l'ocupació de l'hotel. Per determinar-la es vol un llistat amb el nombre d'habitacions ocupades per planta.

```
1 private int[] ocupacio(int[][] hotel){  
2     int [] ocupades = new int[NUM_PLANTES];  
3     int num = 0;  
4     for (int pis = 0; pis < NUM_PLANTES; pis++) {  
5         for (int hab = 0; hab < NUM_HAB; hab++) {  
6             if(hotel[pis][hab] != 0){  
7                 num++;  
8             }  
9         }  
10    }
```

```

10         ocupades[pis] = num;
11         num=0;
12     }
13     return ocupades;
14 }
```

Amb la funció anterior es determina quantes habitacions tenen com a mínim un client a cada planta. Utilitzem la variable *num* com a comptador del nombre d'habitacions plenes. Cada vegada que recorrem una planta tornem a posar a 0 el comptador. Finalment, la funció retorna un *array* on cada posició correspon a la planta, i el valor de la posició al nombre d'habitacions que estan ocupades.

Una manera de llistar l'*array* resultant de l'execució de la funció *ocupació* podria ser:

```

1 //Llistat amb l'ocupació de l'hotel
2 int planta = 0;
3 for (int num : ocupacio(hotel)) {
4     switch(num){
5         case 0:
6             out.println("<li>A la planta " + planta + " estan totes les
7                 habitacions buides</li>");
8             break;
9         case NUM_HAB:
10            out.println("<li>A la planta " + planta + " estan totes les
11                 habitacions plenes.</li>");
12            break;
13        default:
14            out.println("<li>A la planta " + planta + " hi ha " + num + "
15                 habitacions ocupades.</li>");
16     }
17     planta++;
18 }
```

Hem utilitzat l'estructura del bucle *foreach* perquè és molt més neta. Queda molt clar el recorregut que es fa amb l'*array*. Però com que amb aquesta estructura no podem accedir a l'índex de l'element actual a cada volta del bucle, hem de crear-nos el nostre propi índex. Aquest índex només el voldrem per donar la informació de la planta actual a l'usuari.

Un possible resultat de l'execució del codi anterior, quan afegim les funcions anteriors al codi del hotel, seria:

```

1 Gestió de les habitacions d'un hotel
2
3 A la planta 0 estan totes les habitacions buides.
4 A la planta 1 hi ha 9 habitacions ocupades.
5 A la planta 2 hi ha 9 habitacions ocupades.
6 A la planta 3 estan totes les habitacions plenes.
7 A la planta 4 hi ha 8 habitacions ocupades
```

Per acabar, codificarem la funció que determina si existeix alguna habitació buida a l'hotel.

```

1 private boolean habBuides(int[][] hotel) {
2     boolean trobat = false;
3     int pis = 0, porta = 0;
4     while (!trobat && pis < NUM_PLANTES) {
5         if (hotel[pis][porta] == 0) {
6             trobat = true;
```

```

7     }
8     if (porta == NUM_HAB - 1) {
9         porta = 0;
10        pis++;
11    } else {
12        porta++;
13    }
14 }
15 return trobat;
16 }
```

Per mostrar el resultat de la funció anterior a l'usuari s'ha optat per utilitzar un operador ternari dintre de les etiquetes <%= ... %>.

```

1 <h2><%= habBuides(hotel)? "Almenys hi ha una habitació lliure." : "No
   existeixen habitacions lliures." %></h2>
```

La funció per recórrer la matriu és la mateixa que la que hem vist a l'apartat de PHP, únicament s'ha adaptat al llenguatge Java.

Un exemple de l'execució del codi anterior seria el següent:

```

1 Gestió de les habitacions d'un hotel
2
3 A la planta 0 estan totes les habitacions plenes.
4 A la planta 1 hi han 9 habitacions ocupades.
5 A la planta 2 estan totes les habitacions plenes.
6 A la planta 3 hi han 9 habitacions ocupades.
7 A la planta 4 hi han 9 habitacions ocupades.
8 Almenys hi ha una habitació lliure.
```

Ja hem vist com utilitzar una matriu amb la sintaxi estàndard de JSP. Ara veurem la traducció d'alguns exemples a la sintaxi JSTL, i farem el primer exemple. Es vol llistar quants clients hi ha en cada habitació de l'hotel.

```

1 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2 <%@page contentType="text/html" pageEncoding="UTF-8"%>
3 <%
4     private static final int NUM_PLANTES = 5;
5     private static final int NUM_HAB = 10;
6     private static final int MAX_CLIENTS = 5;
7     private int rand(int min, int max) {
8         return (int) (Math.random() * (max + 1 - min) + min);
9     }
10 %
11 <!DOCTYPE html>
12 <html>
13 <head>
14     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
15     <title>Gestió de les habitacions d'un hotel</title>
16 </head>
17 <body>
18     <H1>Gestió de les habitacions d'un hotel</H1>
19     <%
20         //inicialització de les taules amb els comensals
21         int[][] hotel = new int[NUM_PLANTES][NUM_HAB];
22         for (int pis = 0; pis < NUM_PLANTES; pis++) {
23             for (int hab = 0; hab < NUM_HAB; hab++) {
24                 hotel[pis][hab] = rand(0, MAX_CLIENTS);
25             }
26         }
27         request.setAttribute("hotel", hotel);
28         //Llistat amb el nombre de clients per habitacions
29     <%>
```

```

30      %>
31      <ul>
32          <c:forEach var="planta" varStatus="numplanta" items="${hotel}">
33              <c:forEach var="clients" varStatus="numporta" items="${planta}"
34                  ">
35                  <c:out value="${'<li>La habitació'} ${numporta.index} ${'de'
36                      la planta'} ${numplanta.index} ${'té'} ${clients} ${'clients</li>'}" escapeXml="false"></c:out>
37              </c:forEach>
38          </c:forEach>
39      </ul>
    </body>
</html>

```

En aquesta ocasió s'ha enregistrat a l'objecte *request* la matriu bidimensional *hotel*, que està inicialitzada amb tots els clients que hi ha a l'hotel. Es deixa a les etiquetes JSTL el llistat de la informació que conté.

Per poder iterar l'*array* calen dos bucles *foreach*. El primer bucle ens dóna l'*array* corresponent a les habitacions que s'ha de recórrer al bucle següent.

```

1  <c:forEach var="planta" varStatus="numplanta" items="${hotel}">
2  ...
3  </c:forEach>

```

La matriu *hotel* s'assigna a l'atribut *items* de l'etiqueta *foreach*. A cada iteració omplirà la variable *planta* amb l'*array* de clients per habitació que hi ha en aquella planta de l'hotel. A la variable *numplanta* tenim el número d'iteració actual, és a dir, el número de planta actual.

```

1  <c:forEach var="planta" varStatus="numplanta" items="${hotel}">
2      <c:forEach var="clients" varStatus="numporta" items="${planta}">
3          ...
4      </c:forEach>
5  </c:forEach>

```

Així, al bucle intern iterem sobre l'*array* *planta* i accedim al nombre de clients de cada habitació de la planta amb la variable *clients*. La variable *numporta* ens dóna la informació sobre el numero d'habitació actual.

Amb totes aquestes dades ja podem mostrar per pantalla la informació que ens demana l'enunciati:

```

1  <c:out value="${'<li>La habitació'} ${numporta.index} ${'de la planta'} ${numplanta.index} ${'té'} ${clients} ${'clients</li>'}" escapeXml="false"
2  ></c:out>

```

Utilitzant l'etiqueta *<c:out* i amb expressions *EL* podem enviar al navegador la informació demandada. El resultat de l'execució del codi anterior seria el següent:

```

1 Gestió de les habitacions d'un hotel
2
3 L'habitació 0 de la planta 0 té 4 clients
4 L'habitació 1 de la planta 0 té 0 clients
5 L'habitació 2 de la planta 0 té 1 clients
6 L'habitació 3 de la planta 0 té 3 clients
7 L'habitació 4 de la planta 0 té 0 clients
8 L'habitació 5 de la planta 0 té 3 clients
9 L'habitació 6 de la planta 0 té 2 clients

```

```
10 L'habitació 7 de la planta 0 té 3 clients
11 L'habitació 8 de la planta 0 té 4 clients
12 L'habitació 9 de la planta 0 té 3 clients
13 ...
14 L'habitació 0 de la planta 4 té 0 clients
15 L'habitació 1 de la planta 4 té 3 clients
16 L'habitació 2 de la planta 4 té 2 clients
17 L'habitació 3 de la planta 4 té 5 clients
18 L'habitació 4 de la planta 4 té 4 clients
19 L'habitació 5 de la planta 4 té 2 clients
20 L'habitació 6 de la planta 4 té 4 clients
21 L'habitació 7 de la planta 4 té 1 clients
22 L'habitació 8 de la planta 4 té 4 clients
23 L'habitació 9 de la planta 4 té 5 clients
```

Intenteu traduir a JSTL, dintre de les possibilitats del llenguatge, l'ocupació de l'hotel tal com hem fet anteriorment. En concret, volem un llistat amb l'ocupació de les habitacions de l'hotel i una funció que ens indiqui si hi ha habitacions lliures.

2.5 Què s'ha après?

En aquest apartat l'alumne ha après:

- Les nocions bàsiques dels llenguatges: PHP, JSP i JSTL.
- Crear petites aplicacions amb els llenguatges anteriors.
- Utilització dels *arrays* per emmagatzemar un conjunt de dades.
- Crear i utilitzar matrius bidimensionals.

Per aprofundir en aquests llenguatges es recomana la realització de les activitats associades a aquest apartat. Una vegada fetes, l'alumne estarà preparat per endinsar-se en la programació amb Java Servlets.

Desenvolupament web en entorn servidor

Àlex Salinas Tejedor

Desenvolupament web en entorn servidor

Índex

Introducció	5
Resultats d'aprenentatge	7
1 'Servlets'	9
1.1 'Servlet' Hola, Món	9
1.2 'Servlet' EndevinaColor	16
1.3 'Servlet' Publicitat als nous	20
1.3.1 Funcionament d'un 'servlet'. Cicle de vida	23
1.3.2 Exemple de Publicitat amb redireccions	27
1.3.3 Exemple EndevinaColor	29
1.4 Què s'ha après	31
2 Formularis amb 'servlets' i EJB	33
2.1 Calculant el sou net	33
2.2 Escrivint un 'post'	36
2.2.1 'Beans' de sessió amb estat	47
2.3 Dades de subscripció	53
2.4 Què s'ha après?	67
3 Manteniment d'estat, autenticació i autorització amb 'servlets' i EJB	69
3.1 L'usuari, en una galeta	69
3.1.1 Altres maneres d'enviar informació al client	76
3.2 L'usuari a la sessió	79
3.3 Un formulari d'autenticació amb EJB	82
3.4 Què s'ha après?	91

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'introduceix 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 embedguts 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 embedgudes 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.

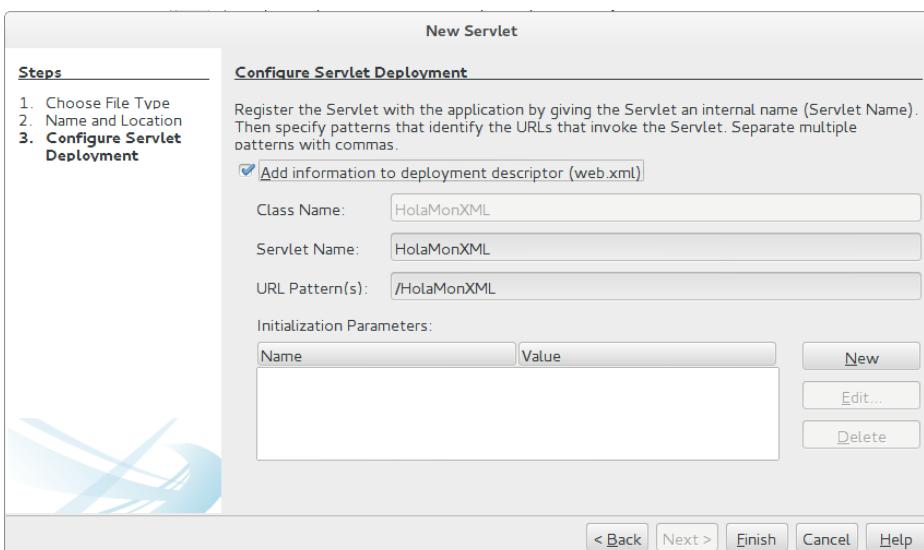
Tots els conceptes que es veuran s'explicaran sempre partint de l'exemple. Quan acabeu aquest apartat estareu preparats per crear un *servlet*, analitzar les peticions i crear les respostes adequades.

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.

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'



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.

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,
26                                   HttpServletResponse response)
27             throws ServletException, IOException {
28         response.setContentType("text/html; charset=UTF-8");
29         try (PrintWriter out = response.getWriter()) {
30             /* TODO output your page here. You may use following sample code.
31             */
32             out.println("<!DOCTYPE html>");
33             out.println("<html>");
34             out.println("<head>");
35             out.println("<title>Servlet HolaMon</title>");
36             out.println("</head>");
37             out.println("<body>");
38             out.println("<h1>Servlet HolaMon at " + request.getContextPath() +
39                         "</h1>");
40             out.println("</body>");
41             out.println("</html>");
42         }
43     }
44
45     // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click
46     // on the + sign on the left to edit the code.">
47     /**
48      * Handles the HTTP GET 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 doGet(HttpServletRequest request, HttpServletResponse
57                           response)
58             throws ServletException, IOException {
59         processRequest(request, response);
60     }
61 }
```

```

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
67                           response)
68         throws ServletException, IOException {
69     processRequest(request, response);
70 }
71
72     /**
73      * Returns a short description of the servlet.
74      *
75      * @return a String containing servlet description
76      */
77     @Override
78     public String getServletInfo() {
79         return "Short description";
80     } // </editor-fold>
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,
24                                   HttpServletResponse response)
25         throws ServletException, IOException {
26     response.setContentType("text/html;charset=UTF-8");
27     try (PrintWriter out = response.getWriter()) {
28         /* TODO output your page here. You may use following sample code.
29         */
30         out.println("<!DOCTYPE html>");
31         out.println("<html>");
32         out.println("<head>");
33         out.println("<title>Servlet HolaMonXML</title>");
34         out.println("</head>");
35         out.println("<body>");
36         out.println("<h1>Servlet HolaMonXML at " + request.getContextPath()
37                     + "</h1>");
38         out.println("</body>");
```

```

36             out.println("</html>");
37         }
38     }
39
40 // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click
41 // 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
52                     response)
53                     throws ServletException, IOException {
54     processRequest(request, response);
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
66                     response)
67                     throws ServletException, IOException {
68     processRequest(request, response);
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 }

```

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.

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

```
1 protected void doGet(HttpServletRequest request, HttpServletResponse response)
2     throws ServletException, IOException {
3     processRequest(request, response);
4 }
```

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.

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

```
1 protected void doPost(HttpServletRequest request, HttpServletResponse response)
2     throws ServletException, IOException {
3     processRequest(request, response);
4 }
```

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:

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

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

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

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

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

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:

```

1 @WebServlet(
2     name = "ServletExemple",
3     description = "Un exemple d'anotació Servlet",
4     urlPatterns = {"/ServletExemple"}
5 )
6 public class ServletExemple extends HttpServlet {
7     // codi...
8 }
9
10 //o amb més d'una URL
11 @WebServlet(
12     urlPatterns = {"/foo", "/bar", "/cool"}
13 )
14 public class ServletExemple extends HttpServlet {
15     // codi...
16 }
```

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.

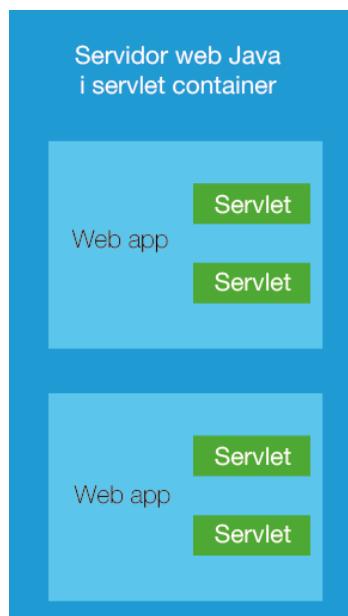
En general:

```
1 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'accendir 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");
```

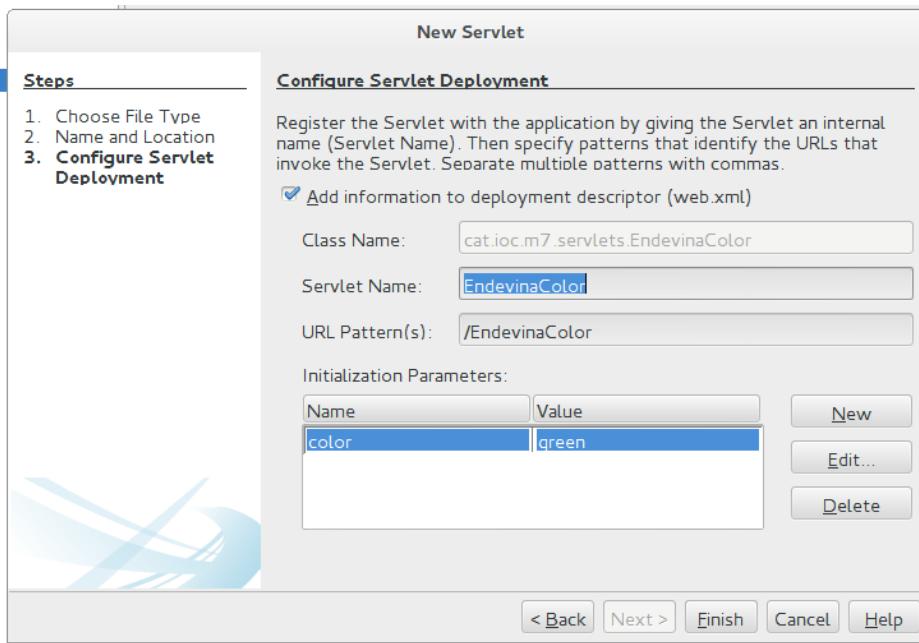
Exemple de lectura d'un atribut en el *ServletContext*:

```
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">
...
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>
...
18 </web-app>
```

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

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

A Java EE existeixen dos tipus de descriptors de desplegament (*deployment 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.

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:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Start Page</title>
5          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6      </head>
7      <body>
8          <h1>Endevina el color configurat:</h1>
9          <a href='EndevinaColor?color=white'>branc</a>
10         <a href='EndevinaColor?color=red'>vermell</a>
11         <a href='EndevinaColor?color=blue'>blau</a>
12         <a href='EndevinaColor?color=yellow'>groc</a>
13         <a href='EndevinaColor?color=green'>verd</a>
14         <a href='EndevinaColor?color=black'>negre</a>
15     </body>
16 </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:

```

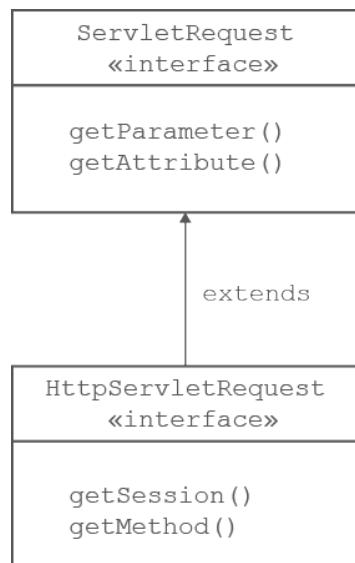
1  protected void processRequest(HttpServletRequest request, HttpServletResponse
2      response) throws ServletException, IOException {
3      response.setContentType("text/html;charset=UTF-8");
4      try (PrintWriter out = response.getWriter()) {
5
6          out.println("<!DOCTYPE html>");
7          out.println("<html>");
8          out.println("<head>");
9          out.println("<title>Endevina el color</title>");
10         out.println("</head>");
11         out.println("<body>");
12
13         String endevinat = "Llàstima, has perdut!";
14
15         //L'usuari ha seleccionat un color i ho ha enviat.
16         String colorUsuari = request.getParameter("color");
17
18         //S'ha configurat un paràmetre que conté el color a endevinar:
19         String colorInicial = getServletConfig().getInitParameter("color");
20
21         if(colorInicial.toLowerCase().equals(colorUsuari.toLowerCase())){
22             endevinat = "Felicitats! Has endevinat el color.";
23         }
24
25         out.println("<h1>" + endevinat +"</h1>");
26         out.println("<a href='EndevinaColor.html'>Tornar a intentar</a>");
27         out.println("</body>");
28         out.println("</html>");
29     }
30 }
```

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.

```

1 String endevinat = "Llàstima, has perdut!";
2 ...
3 if(colorInicial.toLowerCase().equals(colorUsuari.toLowerCase())){
4     endevinat = "Felicitats! Has endevinat el color.";
5 }
6
7 out.println("<h1>" + endevinat + "</h1>");
```

Utilitzant la variable *endevinat* mostrem la informació a l’usuari. En el cas que sigui el mateix color es mostrerà la frase “Felicitats! Has endevinat el color”; en cas contrari es mostrerà “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 mostrerà una pàgina amb un text publicitari; si no és la primera vegada se’n mostrerà 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:

```

1 public class Publicitat extends HttpServlet {
2     private int visites;
3
4     private int num;
5     private String urlPublicitat;
6     private HashMap ip;
7
8     @Override
9     public void init (ServletConfig config) throws ServletException
10    {
11        super.init(config);
12        this.ip = new HashMap();
13        this.num++;
```

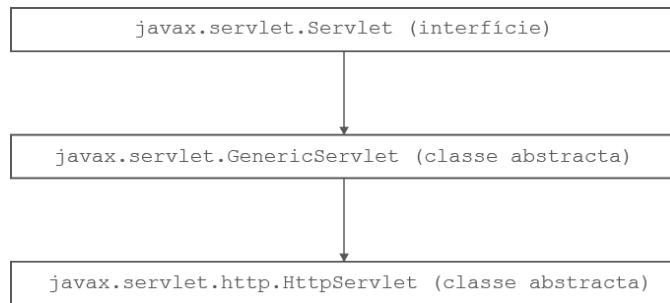
```

14     this.visites = 0;
15     this.urlPublicitat = getServletConfig().getInitParameter("url");
16 }
17
18 @Override
19 public void service(HttpServletRequest request, HttpServletResponse
20     response) throws ServletException, IOException{
21     response.setContentType("text/html;charset=UTF-8");
22     try (PrintWriter out = response.getWriter()) {
23
24         out.println("<!DOCTYPE html>");
25         out.println("<html>");
26         out.println("<head>");
27         out.println("<title>Servlet Publicitat</title>");
28         out.println("</head>");
29         out.println("<body>");
30
31         String requestIp = request.getRemoteAddr();
32
33         if(this.ip.containsKey(requestIp)){
34
35             noEsLaPrimeraVegada(out);
36         }
37         else{
38
39             esLaPrimeraVegada(requestIp, out);
40         }
41
42         out.println("<h5>S'han fet " + this.visites +" visites</h5>");
43         out.println("<h5>S'ha cridat el mètode init " + this.num +" vegades
44             </h5>");
45         out.println("</body>");
46         out.println("</html>");
47
48         this.visites++;
49     }
50 }
51
52 private void noEsLaPrimeraVegada(PrintWriter out){
53     out.println("<h1>Gràcies per tornar a la pàgina web. Ja no veuràs el
54         patrocinador.</h1>");
55 }
56
57 private void esLaPrimeraVegada(String requestIp, PrintWriter out){
58
59     this.ip.put(requestIp, "");
60
61     out.println("<h1>És la primera vegada que accedeixes a la pàgina.
62         Benvingut.</h1>");
63     out.println("<p style='color:red;'>Accedeix al nostre patrocinador
64         clicant al següent enllaç:</p>");
65     out.println("<a href='" + this.urlPublicitat + "'>Pàgina web del
66         patrocinador</a>");
67 }
68 }
```

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'

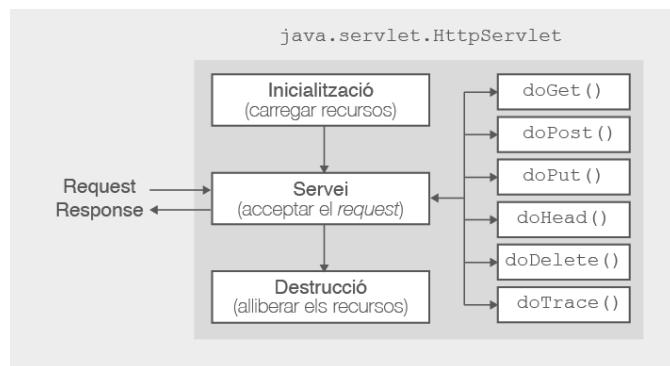
La interfície *servlet* és la interfície més genèrica. Tots els mètodes d'aquesta interfície s'han d'implementar. Els mètodes són: *init*, *service*, *destroy*, *getServletInfo* i *getServletConfig*.

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 sobreescrueix.

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 *HttpServletRequest* 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     } else if (method.equals(METHOD_PUT)) {
11         doPut(req, resp);
12     }
  
```

```

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

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

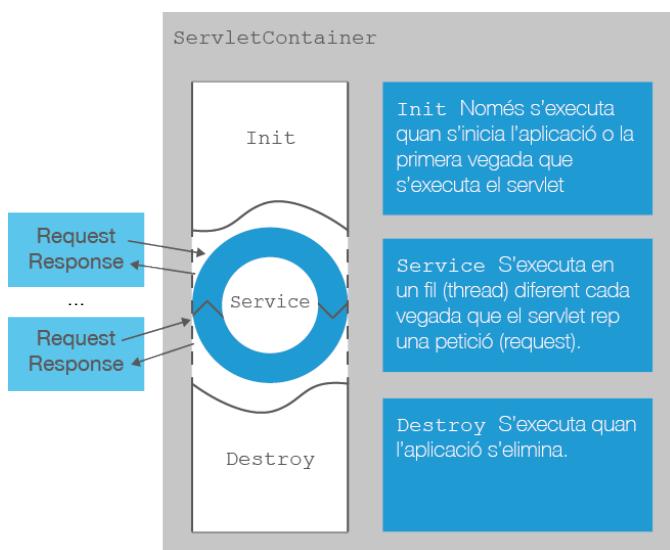
1.3.1 Funcionament d'un 'servlet'. Cicle de vida

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).

FIGURA 1.7. Cicle de vida d'un 'servlet'



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 són les aplicacions gràfiques i les aplicacions basades en client-servidor.

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 base de 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 dels 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:

```

1 public void service(HttpServletRequest request, HttpServletResponse response)
2     throws ServletException, IOException{
3     response.setContentType("text/html;charset=UTF-8");
4     try (PrintWriter out = response.getWriter()) {
5
6         out.println("<!DOCTYPE html>");
7         out.println("<html>");
8         out.println("<head>");
9         out.println("<title>Servlet Publicitat</title>");
10        out.println("</head>");
11        out.println("<body>");
12
13        String requestIp = request.getRemoteAddr();
14
15        if(this.ip.containsKey(requestIp)){
16
17            noEsLaPrimeraVegada(out);
18        }
19        else{
20
21            esLaPrimeraVegada(requestIp, out);
22
23            out.println("<h5>S'han fet " + this.visites +" visites</h5>");
24            out.println("<h5>S'ha cridat el mètode init " + this.num +" vegades
25                </h5>");
26            out.println("</body>");
27            out.println("</html>");
28
29            this.visites++;
30        }
31    }
32 }
```

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:

```

1 String requestIp = request.getRemoteAddr();
2
3 if(this.ip.containsKey(requestIp)){
4
5     noEsLaPrimeraVegada(out);
6 }
7 else{
8
9     esLaPrimeraVegada(requestIp, out);
10 }
```

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

```
1 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
6          .</h1>");
7      out.println("<p style='color:red;'>Accedeix al nostre patrocinador clicant
8          al següent enllaç:</p>");
9      out.println("<a href='" + this.urlPublicitat + "'>Pàgina web del
10         patrocinador</a>");
11 }
```

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'executarà 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
3          Patrocinador.</h1>");
4 }
```

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:

```
1 public void destroy () {
2     // La finalització de codi ...
3 }
```

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:

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Servlet Publicitat</title>
5         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     </head>
7     <body>
8         <h1>És la primera vegada que accedeixes a la pàgina. Benvingut.</h1>
9         <p style='color:red;'>Accedeix al nostre patrocinador clicant al següent enllaç:</p>
10        <a href='http://ioc.xtec.cat'>Pàgina web del patrocinador</a><br>
11        <a href='Publicitat2'>Tornar a accedir</a>
12    </body>
13 </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:

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Servlet Publicitat</title>
5         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     </head>
7     <body>
8         <h1>Gràcies per tornar a la pàgina web. Ja no veuràs el patrocinador.</h1>
9         <a href='Publicitat2'>Tornar a accedir</a>
10    </body>
11 </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:

```

1 ...
2 <servlet>
3   <servlet-name>Publicitat2</servlet-name>
4   <servlet-class>cat.ioc.m7.servlets.Publicitat2</servlet-class>
5 </servlet>
6 <servlet-mapping>
7   <servlet-name>Publicitat2</servlet-name>
8   <url-pattern>/Publicitat2</url-pattern>
9 </servlet-mapping>
10 ...

```

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

```

1 public class Publicitat2 extends HttpServlet {
2
3     private HashMap ip;
4
5     @Override
6     public void init (ServletConfig config) throws ServletException
7     {
8         super.init(config);
9         this.ip = new HashMap();
10    }
11
12    @Override
13    public void service(HttpServletRequest request, HttpServletResponse
14                      response) throws ServletException, IOException{
15
16        String requestIp = request.getRemoteAddr();
17
18        if(this.ip.containsKey(requestIp)){
19
20            //redirect html – resta de vegades
21            response.sendRedirect("resta_vegades.html");
22        }
23        else{
24
25            //redirectHTML – la vegada
26            response.sendRedirect("la_vegada.html");
27            this.ip.put(requestIp, "");
28        }
29    }
30 }

```

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 dependent 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 redireccionalitzar 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 canviará.

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

```
1 RequestDispatcher rs = request.getRequestDispatcher("nouRecurs.html");
2 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 sobreescrivir el mètode `service` pots utilitzar el mètode `processRequest` que crea l'IDE NetBeans.

```
1 private String colorInicial;
2
3 @Override
4 public void init (ServletConfig config) throws ServletException
5 {
6
7     super.init(config);
8
9     //Color configurat com a paràmetre inicial. És el color a endevinar.
10    this.colorInicial = getServletConfig().getInitParameter("color");
11 }
12
13
14 protected void processRequest(HttpServletRequest request, HttpServletResponse
15     response)
16         throws ServletException, IOException {
17
18     response.setContentType("text/html;charset=UTF-8");
19     try (PrintWriter out = response.getWriter()) {
20
21         out.println("<!DOCTYPE html>");
22         out.println("<html>");
23         out.println("<head>");
24         out.println("<title>Endevina el color</title>");
25         out.println("</head>");
26         out.println("<body>");
27
28         String colorUsuari = request.getParameter("color");
```

```

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

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 dependent 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 perdit!";
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 dependent 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 introduïrem 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*.

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'accendir 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.

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.

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 envii 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:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>form-servlet</title>
5          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6      </head>
7      <body>
8          <h1>Càlcul del salari:</h1>
9          <form method="GET" action="CalculSouServlet">
10             <label for="salari">Salari brut:</label>
11             <input id="salari" type="number" name="salariBrut" min="0"/>
12             <label for="fill">Nombre de fills:</label>
13             <input id="fill" type="number" name="fills" min="0">
14             <input type="submit">
15         </form>
16     </body>
17 </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:

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

Aquest camp del formulari és de tipus numèric, i com a mínim el valor que introduceixi l'usuari ha de ser 0, així es fa un petit control d'errors abans d'enviar qualsevol dada al *servlet*. Quan l'usuari introduceixi 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:

```
1  <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 introduceixi 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.

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

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

```
1 @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 calculareu 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:

```
1 protected void processRequest(HttpServletRequest request, HttpServletResponse
2     response)
3     throws ServletException, IOException {
4     response.setContentType("text/html;charset=UTF-8");
5     try (PrintWriter out = response.getWriter()) {
6         out.println("<!DOCTYPE html>");
7         out.println("<html>");
8         out.println("<head>");
9         out.println("<title>Servlet formServlet</title>");
10        out.println("</head>");
11        out.println("<body>");
12        out.println("<h1>Dades rebudes del formulari</h1>");
13
14        int brut = Integer.parseInt(request.getParameter("salariBrut"));
15        out.println("<p>Salari brut: " + brut + "</p>");
16
17        int fills = Integer.parseInt(request.getParameter("fills"));
18        out.println("<p>Nombre de fills: " + fills + "</p>");
19
20        out.println("<h1>Càlcul del salari net:</h1>");
21        int retencio = 21 - (5*fills);
22
23        // Fórmula per calcular la retenció: k = (int)(value*(percentage/100.0f
24        //));
25        int net = brut -((int)(brut * (retencio/100.0f)));
26
27        out.println("<p>Tens una retenció del " + retencio + " per cent</p>");
28        out.println("<p>El teu salari net és de " + net + " euros</p>");
29        out.println("<a href='calcularSou.html'> Tornar </a>");
30        out.println("</body>");
31        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:

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

Tota la informació que s'envia des del navegador cap al *servlet* s'emmaga-zA-Z 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:

```
1 <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:

```
1 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 `getParameter` és de tipus `string`.

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

```
1 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.

```
1 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: $retenció = retencióMaxima - (numFills * 5)$ obtenuïu 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:

```
1 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:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>EJB-SERVLET</title>
5          <meta charset="UTF-8">
6          <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      </head>
8      <body>
9          <h1>Escriu un missatge:</h1>
10         <form id="formulari" method="POST" action="PostServlet">
11             <label for="mail">Correu electrònic:</label>
12             <input id="mail" type="text" name="mail"/>
13             <label for="age">Edat:</label>
14             <input id="age" type="number" name="edad" min="0">
15             <input type="submit">
16         </form>
17         <label>Missatge:</label>
18         <textarea name="missatgePost" form="formulari"></textarea>
19     </body>
20 </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=edad*)
- 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'applicacions 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 *frameworks* faciliten al programador la seva tasca donant-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 són els següents: JSP, JSF, Spring, Struts, Tapestry, Google Web Toolkit (GWT), entre d'altres.

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.

Així, necessiteu crear un *servlet* que rebi les dades enviades amb el formulari i que després les envii 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,
8          HttpServletResponse response)
9          throws ServletException, IOException {
10         response.setContentType("text/html;charset=UTF-8");
11         try (PrintWriter out = response.getWriter()) {
12             out.println("<!DOCTYPE html>");
13             out.println("<html>");
14             out.println("<head>");
15             out.println("<title>Servlet ServletValidation</title>");
16             out.println("</head>");
17             out.println("<body>");
18             out.println("<h1>EJB validation</h1>");
19
20             String mail = request.getParameter("mail");
21             if (validation.isValidEmail(mail)) {
22                 out.println("<p>El correu electrònic " + mail + " és vàlid</p>");
23             } else {
24
25
26
27
28
29
2
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
25
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
27
280
281
282
283
284
285
286
287
288
289
28
290
291
292
293
294
295
296
297
298
299
29
2
300
2
301
2
302
2
303
2
304
2
305
2
306
2
307
2
308
2
309
2
30
310
2
311
2
312
2
313
2
314
2
315
2
316
2
317
2
318
2
319
2
31
320
2
321
2
322
2
323
2
324
2
325
2
326
2
327
2
328
2
329
2
32
330
2
331
2
332
2
333
2
334
2
335
2
336
2
337
2
338
2
339
2
33
340
2
341
2
342
2
343
2
344
2
345
2
346
2
347
2
348
2
349
2
34
350
2
351
2
352
2
353
2
354
2
355
2
356
2
357
2
358
2
359
2
35
360
2
361
2
362
2
363
2
364
2
365
2
366
2
367
2
368
2
369
2
36
370
2
371
2
372
2
373
2
374
2
375
2
376
2
377
2
378
2
379
2
37
380
2
381
2
382
2
383
2
384
2
385
2
386
2
387
2
388
2
389
2
38
390
2
391
2
392
2
393
2
394
2
395
2
396
2
397
2
398
2
399
2
39
3
400
2
3
401
2
3
402
2
3
403
2
3
404
2
3
405
2
3
406
2
3
407
2
3
408
2
3
409
2
3
40
410
2
411
2
412
2
413
2
414
2
415
2
416
2
417
2
418
2
419
2
41
420
2
421
2
422
2
423
2
424
2
425
2
426
2
427
2
428
2
429
2
42
430
2
431
2
432
2
433
2
434
2
435
2
436
2
437
2
438
2
439
2
43
440
2
441
2
442
2
443
2
444
2
445
2
446
2
447
2
448
2
449
2
44
450
2
451
2
452
2
453
2
454
2
455
2
456
2
457
2
458
2
459
2
45
460
2
461
2
462
2
463
2
464
2
465
2
466
2
467
2
468
2
469
2
46
470
2
471
2
472
2
473
2
474
2
475
2
476
2
477
2
478
2
479
2
47
480
2
481
2
482
2
483
2
484
2
485
2
486
2
487
2
488
2
489
2
48
490
2
491
2
492
2
493
2
494
2
495
2
496
2
497
2
498
2
49
499
2
500
2
501
2
502
2
503
2
504
2
505
2
506
2
507
2
508
2
509
2
50
510
2
511
2
512
2
513
2
514
2
515
2
516
2
517
2
518
2
519
2
51
520
2
521
2
522
2
523
2
524
2
525
2
526
2
527
2
528
2
529
2
52
530
2
531
2
532
2
533
2
534
2
535
2
536
2
537
2
538
2
539
2
53
540
2
541
2
542
2
543
2
544
2
545
2
546
2
547
2
548
2
549
2
54
550
2
551
2
552
2
553
2
554
2
555
2
556
2
557
2
558
2
559
2
55
560
2
561
2
562
2
563
2
564
2
565
2
566
2
567
2
568
2
569
2
56
570
2
571
2
572
2
573
2
574
2
575
2
576
2
577
2
578
2
579
2
57
580
2
581
2
582
2
583
2
584
2
585
2
586
2
587
2
588
2
589
2
58
590
2
591
2
592
2
593
2
594
2
595
2
596
2
597
2
598
2
59
599
2
600
2
601
2
602
2
603
2
604
2
605
2
606
2
607
2
608
2
609
2
60
610
2
611
2
612
2
613
2
614
2
615
2
616
2
617
2
618
2
619
2
61
620
2
621
2
622
2
623
2
624
2
625
2
626
2
627
2
628
2
629
2
62
630
2
631
2
632
2
633
2
634
2
635
2
636
2
637
2
638
2
639
2
63
640
2
641
2
642
2
643
2
644
2
645
2
646
2
647
2
648
2
649
2
64
650
2
651
2
652
2
653
2
654
2
655
2
656
2
657
2
658
2
659
2
65
660
2
661
2
662
2
663
2
664
2
665
2
666
2
667
2
668
2
669
2
66
670
2
671
2
672
2
673
2
674
2
675
2
676
2
677
2
678
2
679
2
67
680
2
681
2
682
2
683
2
684
2
685
2
686
2
687
2
688
2
689
2
68
690
2
691
2
692
2
693
2
694
2
695
2
696
2
697
2
698
2
69
699
2
700
2
701
2
702
2
703
2
704
2
705
2
706
2
707
2
708
2
709
2
70
710
2
711
2
712
2
713
2
714
2
715
2
716
2
717
2
718
2
719
2
71
720
2
721
2
722
2
723
2
724
2
725
2
726
2
727
2
728
2
729
2
72
730
2
731
2
732
2
733
2
734
2
735
2
736
2
737
2
738
2
739
2
73
740
2
741
2
742
2
743
2
744
2
745
2
746
2
747
2
748
2
749
2
74
750
2
751
2
752
2
753
2
754
2
755
2
756
2
757
2
758
2
759
2
75
760
2
761
2
762
2
763
2
764
2
765
2
766
2
767
2
768
2
769
2
76
770
2
771
2
772
2
773
2
774
2
775
2
776
2
777
2
778
2
779
2
77
780
2
781
2
782
2
783
2
784
2
785
2
786
2
787
2
788
2
789
2
78
790
2
791
2
792
2
793
2
794
2
795
2
796
2
797
2
798
2
799
2
79
7
800
2
801
2
802
2
803
2
804
2
805
2
806
2
807
2
808
2
809
2
80
810
2
811
2
812
2
813
2
814
2
815
2
816
2
817
2
818
2
819
2
81
820
2
821
2
822
2
823
2
824
2
825
2
826
2
827
2
828
2
829
2
82
830
2
831
2
832
2
833
2
834
2
835
2
836
2
837
2
838
2
839
2
83
840
2
841
2
842
2
843
2
844
2
845
2
846
2
847
2
848
2
849
2
84
850
2
851
2
852
2
853
2
854
2
855
2
856
2
857
2
858
2
859
2
85
860
2
861
2
862
2
863
2
864
2
865
2
866
2
867
2
868
2
869
2
86
870
2
871
2
872
2
873
2
874
2
875
2
876
2
877
2
878
2
879
2
87
880
2
881
2
882
2
883
2
884
2
885
2
886
2
887
2
888
2
889
2
88
890
2
891
2
892
2
893
2
894
2
895
2
896
2
897
2
898
2
899
2
89
8
900
2
901
2
902
2
903
2
904
2
905
2
906
2
907
2
908
2
909
2
90
910
2
911
2
912
2
913
2
914
2
915
2
916
2
917
2
918
2
919
2
91
920
2
921
2
922
2
923
2
924
2
925
2
926
2
927
2
928
2
929
2
92
930
2
931
2
932
2
933
2
934
2
935
2
936
2
937
2
938
2
939
2
93
940
2
941
2
942
2
943
2
944
2
945
2
946
2
947
2
948
2
949
2
94
950
2
951
2
952
2
953
2
954
2
955
2
956
2
957
2
958
2
959
2
95
960
2
961
2
962
2
963
2
964
2
965
2
966
2
967
2
968
2
969
2
96
970
2
971
2
972
2
973
2
974
2
975
2
976
2
977
2
978
2
979
2
97
980
2
981
2
982
2
983
2
984
2
985
2
986
2
987
2
988
2
989
2
98
990
2
991
2
992
2
993
2
994
2
995
2
996
2
997
2
998
2
999
2
99
9

```

```

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
29                     missatge a l'aplicació.</p>");
30     } else {
31         out.println("<p>Has de ser major d'edat per escriure un
32                     missatge a l'aplicació.</p>");
33     }
34
35     String missatge = request.getParameter("missatgePost");
36     if (validation.isValidPost(missatge)) {
37         out.println("<p>El missatge '" + missatge + "' és vàlid.</p>");
38     } else {
39         out.println("<p>El missatge no és vàlid. Ha de tenir menys de
40                     150 caràcters.</p>");
41     }
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:

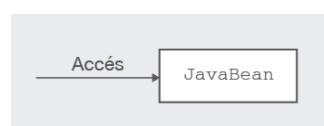
```

1 if (validation.isValidAge(edat)) {
2     out.println("<p>Tens " + edat + " anys i pots escriure un missatge a l'
3                 aplicació.</p>");
4 } else {
5     out.println("<p>Has de ser major d'edat per escriure un missatge a l'
6                 aplicació.</p>");
7 }
```

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.

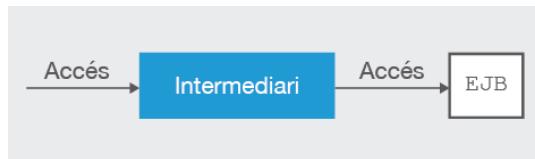
Un EJB (Enterprise JavaBean) és un component que ha d'executar-se en un contenidor d'EJB, i es diferencia molt d'un JavaBean normal. Un JavaBean és un objecte al qual accediu de manera directa des de la vostra aplicació (vegeu la figura 3.1).

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'algún tipus d'intermediari (vegeu la figura 3.2).

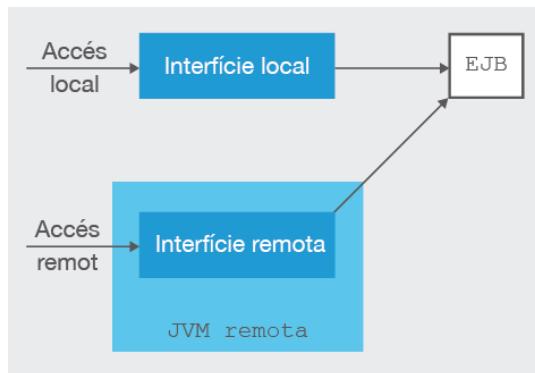
FIGURA 2.2. Accés a un EJB



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.

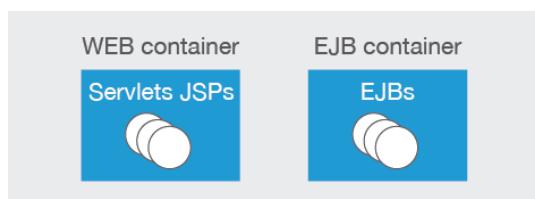
L'intermediari local és aquell que s'executa en la mateixa màquina virtual de Java que l'EJB. En canvi, el remot és aquell intermediari que s'executa en una màquina virtual de Java diferent de la màquina on s'executa l'EJB. Gràcies a aquests intermediaris es poden construir aplicacions distribuïdes (vegeu 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 contingidor EJB (vegeu la figura 3.4), com són el maneig de transaccions, la concorrència, la seguretat, la gestió de recursos, els serveis de xarxa, etc.

FIGURA 2.4. Contingidor d'EJB



Un **contingidor 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*).

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'accendir 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 demandada.

```

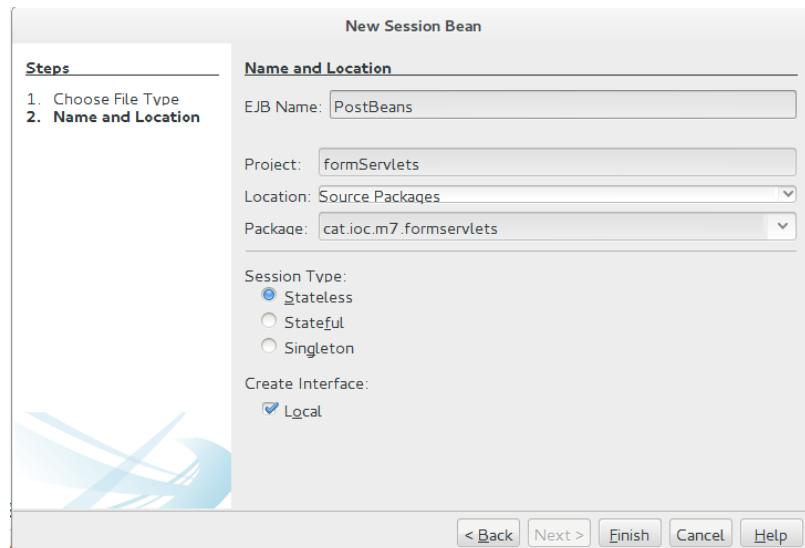
1 if (validation.isValidAge(edat)) {
2     out.println("<p>Tens " + edat + " anys i pots escriure un missatge a l' 
3     aplicació.</p>");
4 } else {
5     out.println("<p>Has de ser major d'edat per escriure un missatge a l' 
6     aplicació.</p>");
7 }
```

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.

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.

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 introduceix 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 sobreescrivir 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-zA-Z]” 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.

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

```

1 String email = "a@a";
2 String regex = "^(.+)@(.+)$";
3 Pattern pattern = Pattern.compile(regex);
4 Matcher matcher = pattern.matcher(email);
5 return matcher.matches();

```

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

```

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,
8         HttpServletResponse response)
9         throws ServletException, IOException {
10        response.setContentType("text/html;charset=UTF-8");
11        try (PrintWriter out = response.getWriter()) {
12            out.println("<!DOCTYPE html>");
13            out.println("<html>");
14            out.println("<head>");
15            out.println("<title>Servlet ServletValidation</title>");
16            out.println("</head>");
17            out.println("<body>");
18            out.println("<h1>EJB validation</h1>");
19
20            String mail = request.getParameter("mail");
21            if (validation.isValidEmail(mail)) {
22                out.println("<p>El correu electrònic " + mail + " és vàlid</p>");
23            } else {
24                out.println("<p>El correu electrònic no és vàlid.</p>");
25            }
26
27            String edat = request.getParameter("edat");
28            if (validation.isValidAge(edat)) {
29                out.println("<p>Tens " + edat + " anys i pots escriure un
30                         missatge a l'aplicació.</p>");
31            } else {
32                out.println("<p>Has de ser major d'edat per escriure un
33                         missatge a l'aplicació.</p>");
34            }
35
36            String missatge = request.getParameter("missatgePost");
37            if (validation.isValidPost(missatge)) {
38                out.println("<p>El missatge '" + missatge + "' és vàlid.</p>");
39            } else {
40                out.println("<p>El missatge no és vàlid. Ha de tenir menys de
41                         150 caràcters.</p>");
42            }
43
44            out.println("</body>");
45            out.println("</html>");
46        }
47    }

```

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*.

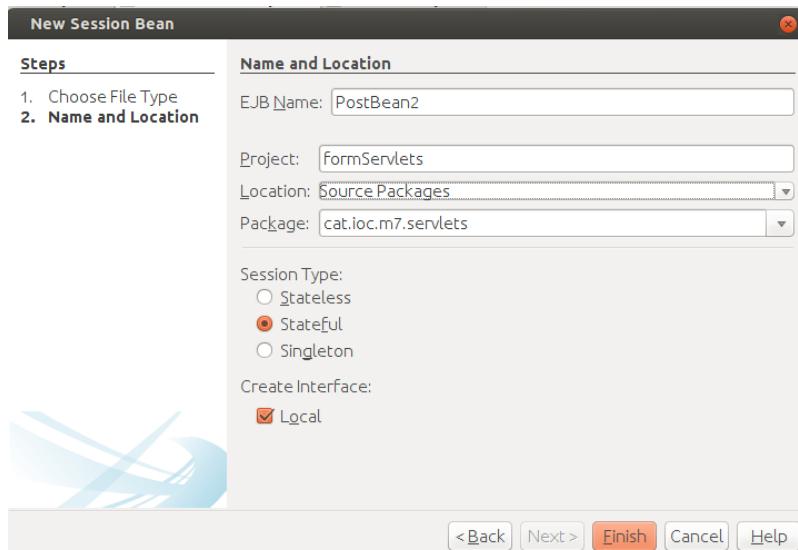
L'objectiu de l'exercici és validar les dades. En comptes d'utilitzar tres mètodes per validar-les emprarem les validacions de Java del *package validation*.

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:

```

1 package cat.ioc.m7.formservlets;
2
3 import javax.ejb.Local;
4
5 @Local
6 public interface PostBean2Local {
7
8     public int getEdat();
9
10    public String getMessage();
11
12    public String getEmail();
13
14    public void setEdat(String edat);
15
16    public void setMessage(String message);
17
18    public void setEmail(String email);
19
20 }
```

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:

```

1 @Stateful
2 public class PostBean2 implements PostBean2Local {
3
4     private int edat;
5
6     private String message;
7
8     private String email;
9
10    @Override
11    public int getEdat() {
12        return edat;
13    }
14
15    @Override
16    public void setEdat(String edat) {
17        if(!edat.equals("")){
18            this.edat=Integer.parseInt(edat);
19        }
20        else{
21            this.edat = 0;
22        }
23    }
24
25    @Override
26    public String getMessage() {
27        return message;
28    }
```

```

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 *edad*, *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'edad per escriure un missatge.
5          ")
6      public int getEdad();
7
8      @NotNull @Size(min=1, max=150, message = "El missatge no és vàlid. Ha de
9          tenir menys de 150 caràcters.")
10     public String getMessage();
11
12     @NotNull @Pattern(regexp="^(.+)@(.+)$", message = "El correu no és vàlid.")
13     public String getEmail();
14
15     public void setEdad(String edad);
16
17     public void setMessage(String message);
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 mostrerà 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,
7          HttpServletResponse response)
8          throws ServletException, IOException {
9
10         response.setContentType("text/html;charset=UTF-8");
11         try (PrintWriter out = response.getWriter()) {
12             out.println("<!DOCTYPE html>");
13             out.println("<html>");
14             out.println("<head>");
15             out.println("<title>Servlet ServletValidation2</title>");
16             out.println("</head>");
17             out.println("<body>");
18
19             String missatge = request.getParameter("missatgePost");
20             String mail = request.getParameter("mail");
21             String edat = request.getParameter("edat");
22
23             PostBean2Local bean = (PostBean2Local) new InitialContext().lookup(
24                 "java:global/formServlets/PostBean2");
25

```

```

26     bean.setMessage(missatge);
27     bean.setEmail(mail);
28     bean.setEdat(edat);
29
30     out.println("<h1>Dades rebudes del formulari</h1>");
31     out.println("<p>Missatge: " + bean.getMessage() + "</p>");
32     out.println("<p>Edat: " + bean.getEdat() + "</p>");
33     out.println("<p>Email: " + bean.getEmail() + "</p>");
34
35     out.println("<h1>Llistat de validacions:</h1>");
36     for (ConstraintViolation c : validator.validate(bean)) {
37         out.println("<p>" + c.getMessage() + "</p>");
38     }
39
40     out.println("</body>");
41     out.println("</html>");
42 } catch (NamingException ex) {
43     Logger.getLogger(PostServlet2.class.getName()).log(Level.SEVERE,
44         null, ex);
45 }
```

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

```

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

i les introduceix en el *bean*:

```

1 PostBean2Local bean = (PostBean2Local) new InitialContext().lookup(
2     "java:global/formServlets/PostBean2");
3
4 bean.setMessage(missatge);
5 bean.setEmail(mail);
6 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.

```

1 PostBean2Local bean = (PostBean2Local) new InitialContext().lookup(
2     "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 contingidor EJB no permet accedir directament a l'objecte.

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

```

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

Una vegada el *bean* té les dades es farà 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.

```

1  for (ConstraintViolation c : validator.validate(bean)) {
2      out.println("<p>" + c.getMessage() + "</p>");
3  }

```

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:

```

1  @Resource Validator validator;

```

L'etiqueta @Resource informa el servidor que l'aplicació necessita un validador. Aquest validador s'injecta, igual que el contingidor 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.

```

1  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:

```

1  for (ConstraintViolation c : validator.validate(bean)) {
2      out.println("<p>" + c.getMessage() + "</p>");
3  }

```

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:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Bean Validation</title>
5          <meta charset="UTF-8">
6          <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      </head>
8      <body>
9          <h1>Escriu un missatge:</h1>
10         <form id="formulari" method="POST" action="PostServlet2">
11             <label for="mail">Correu electrònic:</label>
12             <input id="mail" type="text" name="mail"/>
13             <label for="age">Edat:</label>
14             <input id="age" type="number" name="edad" min="0">
15             <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’anomenieu 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  
27     @Override  
28     public String print() {  
29  
30         return "<p>nom=" + nom + "</p> <p>"  
31         + "cognoms=" + cognoms + "</p><p>"  
32         + "naixement=" + naixement + "</p><p>"  
33         + "sexe=" + sexe + "</p> <p>email=" + email + "</p><p>"  
34         + "telefon=" + telefon + "</p> <p>"  
35         + "color=" + color + "</p> <p>"  
36         + "marcaCotxe=" + marcaCotxe + "</p> <p>"  
37         + "vehicle1=" + vehicle1 + "</p> <p>"  
38         + "vehicle2=" + vehicle2 + "</p> <p>"  
39         + "navegador=" + navegador + "</p>";  
40     }  
41  
42     @Override  
43     public String getNom() {  
44         return nom;  
45     }  
46  
47     @Override  
48     public void setNom(String nom) {  
49         this.nom = nom;  
50     }  
51  
52     @Override  
53     public String getCognoms() {  
54         return cognoms;  
55     }  
56  
57     @Override  
58     public void setCognoms(String cognoms) {  
59         this.cognoms = cognoms;  
60     }  
61  
62     @Override  
63     public String getNaixement() {  
64         return naixement;  
65     }  
66  
67     @Override  
68     public void setNaixement(String naixement) {  
69         this.naixement = naixement;  
70     }  
71  
72     @Override  
73     public String getSexe() {  
74         return sexe;  
75     }  
76  
77     @Override  
78     public void setSexe(String sexe) {  
79         this.sexe = sexe;  
80     }  
81  
82     @Override  
83     public String getEmail() {  
84         return email;  
85     }
```

```
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:

```

1  @Local
2  public interface SignupBeanLocal {
3
4      @NotNull (message = "El nom no pot estar buit.")
5      public String getNom();
6
7      @NotNull(message = "El cognom no pot estar buit.")
8      public String getCognoms();
9
10     public String getNaixement();
11
12     @NotNull(message = "El gènere no pot estar buit.")
13     public String getSexe();
14
15     public String getEmail();
16
17     @Pattern(regexp = "\d{3}--\d{3}-\d{3}", message = "El telèfon no és vàlid.")
18     public String getTelefon();
19
20     public String getColor();
21
22     @NotNull(message = "La marca del cotxe no pot estar buida.")
23     public String getMarcaCotxe();
24
25     public String getVehicle1();
26
27     public String getVehicle2();
28
29     @NotNull(message = "El navegador no pot estar buit.")
30     public String getNavegador();
31
32     public String print();
33
34     public void setNom(String nom);
35
36     public void setCognoms(String cognoms);
37
38     public void setNaixement(String naixement);
39
40     public void setSexe(String sexe);
41
42     public void setEmail(String email);
43
44     public void setTelefon(String telefon);
45
46     public void setColor(String color);
47

```

```

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 dígsits.

```

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:

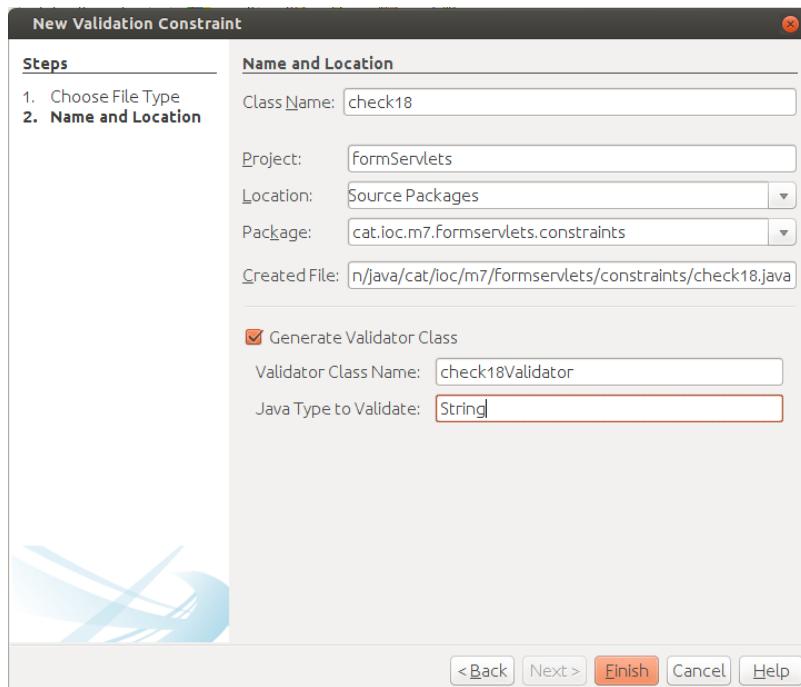
- @Check18: comprova que l'usuari sigui major d'edat. Rep un *string* per paràmetre i comprova que a data d'avui sigui més gran de 18 anys.

- @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.

```

1 @Documented
2 @Constraint(validatedBy = Check18Validator.class)
3 @Target({ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE})
4 @Retention(RetentionPolicy.RUNTIME)
5 public @interface Check18 {
6
7     String message() default "{cat.ioc.m7.formservlets.constraints.PastDate}";
8
9     Class<?>[] groups() default {};
10
11    Class<? extends Payload>[] payload() default {};
12}
```

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'altra 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:

```

1 public class Check18Validator implements ConstraintValidator<Check18, String> {
2
3     @Override
```

```

4   public void initialize(Check18 constraintAnnotation) {
5     }
6
7     @Override
8     public boolean isValid(String value, ConstraintValidatorContext context) {
9       try {
10         if (!value.equals("")) {
11           Date naixement = new SimpleDateFormat("yyyy-MM-dd").parse(value
12             );
13           int anyAra = Calendar.getInstance().get(Calendar.YEAR);
14           Calendar cal = Calendar.getInstance();
15           cal.setTime(naixement);
16           int anyNaixement = cal.get(Calendar.YEAR);
17           return anyAra - anyNaixement >= 18;
18         }
19         catch (ParseException ex) {
20           Logger.getLogger(Check18Validator.class.getName()).log(Level.SEVERE
21             , null, ex);
22         }
23         return false;
24       }
25     }

```

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 sobreescrivir.

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

```

1   if (!value.equals("")) {
2     Date naixement = new SimpleDateFormat("yyyy-MM-dd").parse(value);
3     int anyAra = Calendar.getInstance().get(Calendar.YEAR);
4     Calendar cal = Calendar.getInstance();
5     cal.setTime(naixement);
6     int anyNaixement = cal.get(Calendar.YEAR);
7     return anyAra - anyNaixement >= 18;
8   }
9   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`:

```

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

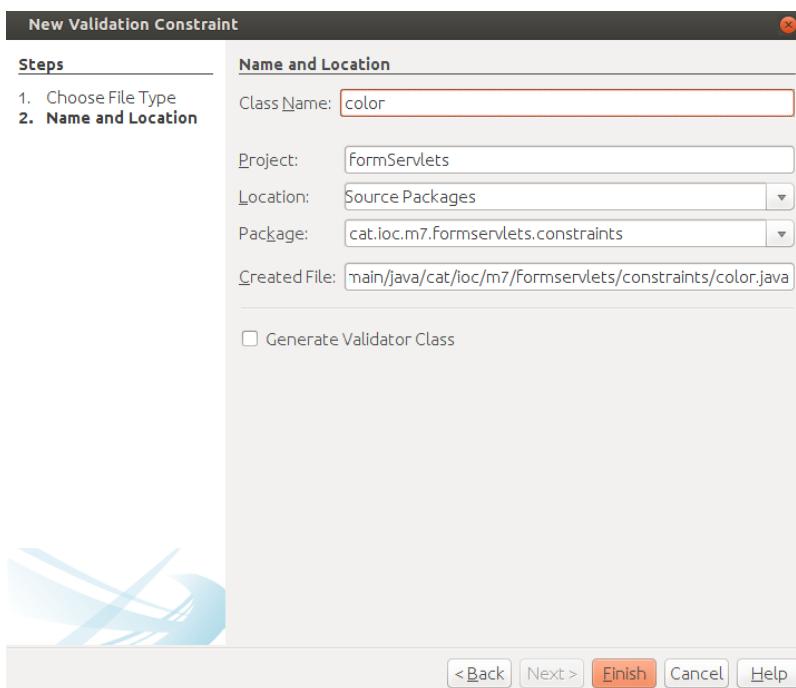
```

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:

```

1 @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:

```

1 @interface List {
2     Color[] value();
3 }

```

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:

```

1  @Pattern.List({@Pattern(regexp = "#([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})")})
2  @Constraint(validatedBy = {})
3  @Documented
4  @Target({ElementType.METHOD,
5      ElementType.FIELD,
6      ElementType.ANNOTATION_TYPE,
7      ElementType.CONSTRUCTOR,
8      ElementType.PARAMETER})
9  @Retention(RetentionPolicy.RUNTIME)
10 public @interface Color {
11
12     String message() default "{invalid.color}";
13
14     Class<?>[] groups() default {};
15
16     Class<? extends Payload>[] payload() default {};
17
18     @Target({ElementType.METHOD,
19         ElementType.FIELD,
20         ElementType.ANNOTATION_TYPE,
21         ElementType.CONSTRUCTOR,
22         ElementType.PARAMETER})
23     @Retention(RetentionPolicy.RUNTIME)
24     @Documented
25     @interface List {
26         Color[] value();
27     }
28 }
```

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.

```

1  @Color(message = "El color no pot estar buit.")
2  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:

```

1  @Local
```

```

2  public interface SignupBeanLocal {
3
4      @NotNull (message = "El nom no pot estar buit.")
5      @NotBlank(message = "El nom no pot estar buit.")
6      public String getNom();
7
8      @NotNull(message = "El cognom no pot estar buit.")
9      @NotBlank(message = "El cognom no pot estar buit.")
10     public String getCognoms();
11
12     @Check18 (message = "Has de tenir més de 18 anys.")
13     @NotBlank(message = "La data de naixement no pot estar buida.")
14     public String getNaixement();
15
16     @NotNull(message = "El gènere no pot estar buit.")
17     public String getSexe();
18
19     @Email
20     @NotBlank(message = "El correu no pot estar buit.")
21     public String getEmail();
22
23     @Pattern-regexp = "\\\(\d{3}\\)\d{3}-\d{4}", message = "El telèfon no és
24         vàlid."
25     @NotBlank(message = "El telèfon no pot estar buit.")
26     public String getTelefon();
27
28     @Color(message = "El color no pot estar buit.")
29     public String getColor();
30
31     @NotNull(message = "La marca del cotxe no pot estar buida.")
32     public String getMarcaCotxe();
33
34     @NotBlank(message = "El vehicle1 no pot estar buit.")
35     public String getVehicle1();
36
37     @NotBlank(message = "El vehicle2 no pot estar buit.")
38     public String getVehicle2();
39
40     @NotNull(message = "El navegador no pot estar buit.")
41     @NotBlank(message = "El navegador no pot estar buit.")
42     public String getNavegador();
43
44     public String print();
45
46     public void setNom(String nom);
47
48     public void setCognoms(String cognoms);
49
50     public void setNaixement(String naixement);
51
52     public void setSexe(String sexe);
53
54     public void setEmail(String email);
55
56     public void setTelefon(String telefon);
57
58     public void setColor(String color);
59
60     public void setMarcaCotxe(String marcaCotxe);
61
62     public void setVehicle1(String vehicle1);
63
64     public void setVehicle2(String vehicle2);
65
66     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:

```

1  @WebServlet(name = "SignUpServlet", urlPatterns = {"/SignUpServlet"})
2  public class SignUpServlet extends HttpServlet {
3
4
5      @Resource Validator validator;
6
7      protected void processRequest(HttpServletRequest request,
8          HttpServletResponse response)
9          throws ServletException, IOException {
10         response.setContentType("text/html;charset=UTF-8");
11         try (PrintWriter out = response.getWriter()) {
12
13             out.println("<!DOCTYPE html>");
14             out.println("<html>");
15             out.println("<head>");
16             out.println("<title>Servlet SingUp</title>");
17             out.println("</head>");
18             out.println("<body>");
19
20             SignupBeanLocal bean = (SignupBeanLocal) new InitialContext().
21                 lookup(
22                     "java:global/formServlets/SignupBean");
23
24             try{
25                 out.println("<h1>Dades rebudes del formulari</h1>");
26                 BeanUtils.populate (bean, request.getParameterMap());
27                 out.println("<p>" + bean.print() + "</p>");
28             }
29             catch(IllegalAccessException | InvocationTargetException e){
30                 out.println(e.getMessage());
31             }
32
33             out.println("<h1>Llistat de validacions:</h1>");
34
35             for (ConstraintViolation c : validator.validate(bean)) {
36                 out.println("<p>" + c.getMessage() + "</p>");
37             }
38
39             out.println("</body>");
40             out.println("</html>");
41         } catch (NamingException ex) {
42             Logger.getLogger(SignUpServlet.class.getName()).log(Level.SEVERE,
43                 null, ex);
44         }
45     }
46 }
```

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* .

```

1  SignupBeanLocal bean = (SignupBeanLocal) new InitialContext().lookup(
2      "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.

```
1 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 corresponguï amb el nom dels mètodes *set* de les propietats. Exemple:

```
1 <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*:

```
1 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*.

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Subscripció</title>
5         <meta charset="UTF-8">
6         <meta name="viewport" content="width=device-width, initial-scale=1.0">
7     </head>
8     <body>
9         <h1>Per subscriure't necessitem les següents dades:</h1>
10        <form id="formulari" method="POST" action="SignUpServlet">
11            <label for="nom">Nom:</label>
12            <input type="text" name="nom" id="nom"><br>
13            <label for="cognoms">Cognoms:</label>
14            <input type="text" name="cognoms" id="cognoms"><br>
15            <label for="naix">Data de naixement:</label>
16            <input id="naix" type="date" name="naixement"><br>
17            <input type="radio" name="sexe" value="home" checked> Home<br>
18            <input type="radio" name="sexe" value="dona"> Dona<br>
19            <label for="email">Correu electrònic:</label>
20            <input id="email" type="text" name="email"><br>
21            <label for="telf">Telèfon:</label>
22            <input id="telf" type="tel" name="telefon"><br>
23            <label for="color">Color preferit:</label>
24            <input id="color" type="color" name="color"><br>
25
26            <label for="cotxe">Marca del cotxe:</label>
27            <select id="cotxe" name="marcaCotxe">
28                <option value="volvo">Volvo</option>
29                <option value="saab">Saab</option>
30                <option value="fiat">Fiat</option>
31                <option value="audi">Audi</option>
32                <option value="audi">Altre</option>
33            </select><br>
34            <input type="checkbox" name="vehicle1" value="Bicicleta"> Tinc una
               bicicleta<br>
```

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.

```

35      <input type="checkbox" name="vehicle2" value="Moto"> Tinc una moto<
36          br>
37      <input list="browsers" name="navegador">
38      <datalist id="browsers">
39          <option value="Internet Explorer">
40          <option value="Firefox">
41          <option value="Chrome">
42          <option value="Opera">
43          <option value="Safari">
44      </datalist>
45      <input type="submit">
46  </form>
47
48  </body>
49
50  </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 voleu. 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 comunique 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 ocells i reescritura d'adreses 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 satisfactoria 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, dependent 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.

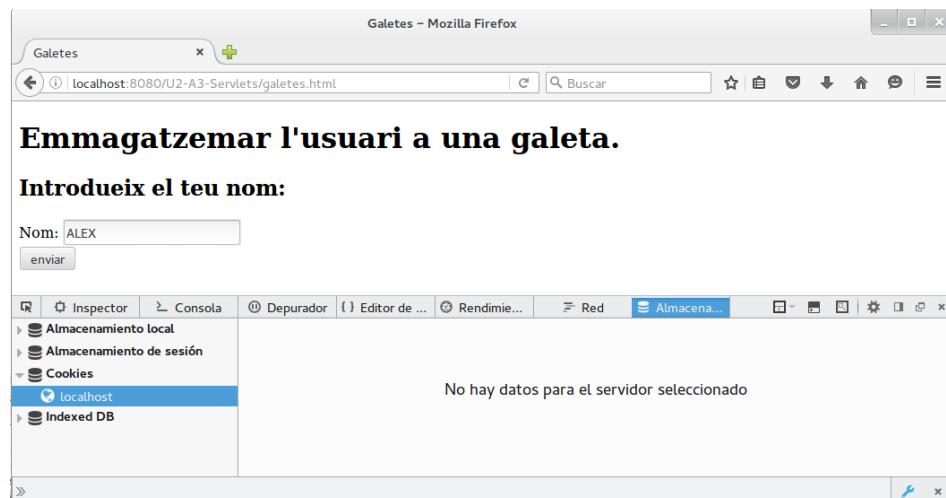
Comencem creant un nou projecte amb Maven, i posteriorment crearem un formulari en una pàgina web que envii les dades d'un usuari a un *servlet* per tal que

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

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Galetes</title>
5         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     </head>
7     <body>
8         <h1>Emmagatzemar l'usuari a una galeta.</h1>
9         <h2>Introdueix el teu nom:</h2>
10        <form action="galetesServlet" method="post">
11            <label for="nom"> Nom:</label>
12            <input id="nom" type="text" name="userName"/> <br/>
13            <input type="submit" value="enviar"/>
14        </form>
15    </body>
16 </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:

```
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
42 // 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
53                     response)
54     throws ServletException, IOException {
55     processRequest(request, response);
56 }
57
58 /**
59 * Handles the HTTP POST method.
60 *
61 * @param request servlet request
62 * @param response servlet response
63 * @throws ServletException if a servlet-specific error occurs
64 * @throws IOException if an I/O error occurs
65 */
66 @Override
67 protected void doPost(HttpServletRequest request, HttpServletResponse
68                     response)
69     throws ServletException, IOException {
70     processRequest(request, response);
71 }
72
73 /**
74 * Returns a short description of the servlet.
75 *
76 * @return a String containing servlet description
77 */
78 @Override
79 public String getServletInfo() {
80     return "Short description";
81 } // </editor-fold>
82 }
```

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](https://)), galetes només http, i inclusí galetes *zombie*. Aquestes últimes s'emmagatzemaren 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(tempo_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
```

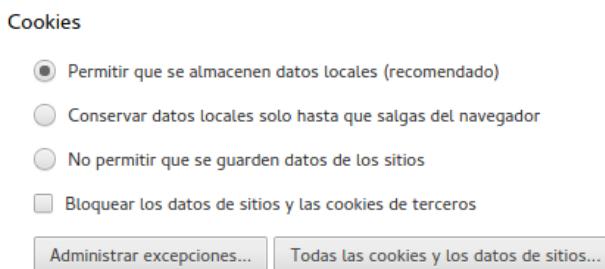
Una vegada s'ha creat la galeta, heu d'enviar-la al navegador perquè aquest sigui qui l'emmagatzemi dins del disc dur client.

```
1 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



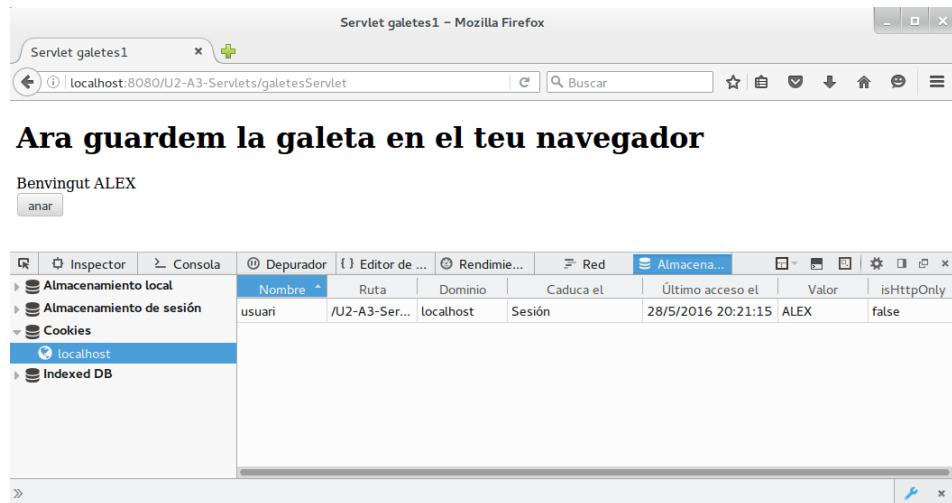
Com heu pogut comprovar, el funcionament de les galetes és senzill:

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 envii 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):

```
1 out.print("<form action='galetesServlet2'>");
2 out.print("<input type='submit' value='anar'>");
3 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,
14         HttpServletResponse response)
15             throws ServletException, IOException {
16         response.setContentType("text/html;charset=UTF-8");
17         try (PrintWriter out = response.getWriter()) {
18             /* TODO output your page here. You may use following sample code.
19             */
20             out.println("<!DOCTYPE html>");
21             out.println("<html>");
22             out.println("<head>");
23             out.println("<title>Servlet galetes2</title>");
24             out.println("</head>");
25             out.println("<body>");
26             out.println("<h1>Accedim a la galeta per veure el nom d'usuari.</h1>");
27             out.println("</body>");
28             out.println("</html>");
29         }
30     }
31
32     // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click
33     // on the + sign on the left to edit the code.">
34     /**
35      * Handles the HTTP GET method.
36      *
37      * @param request servlet request
38      * @param response servlet response
39      */
40 
```

```

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

```

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

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

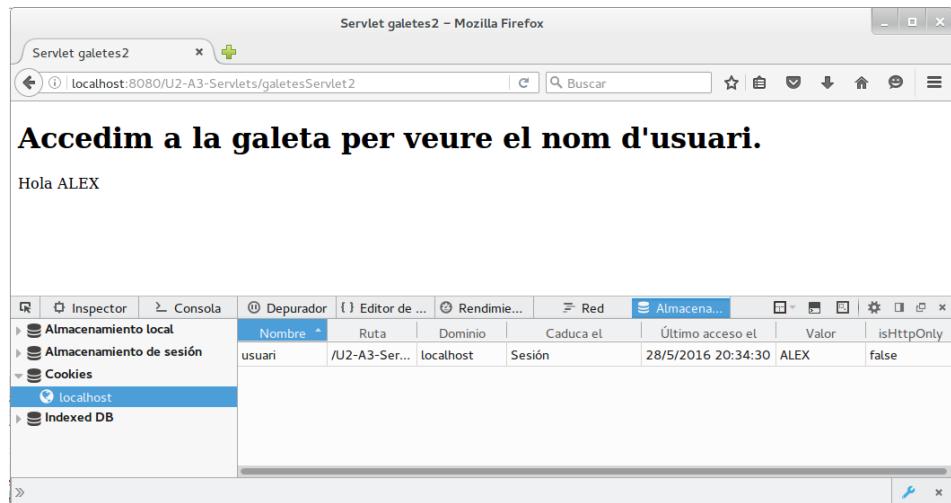
Adoneu-vos que el mètode `getCookies()` retorna totes les galetes que té el navegador de la nostra pàgina web. Així, si teniu més d'una galeta heu de fer un bucle per trobar la que us interessa. En el nostre cas no cal, només n'hem emmagatzemat una.

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

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

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.

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:

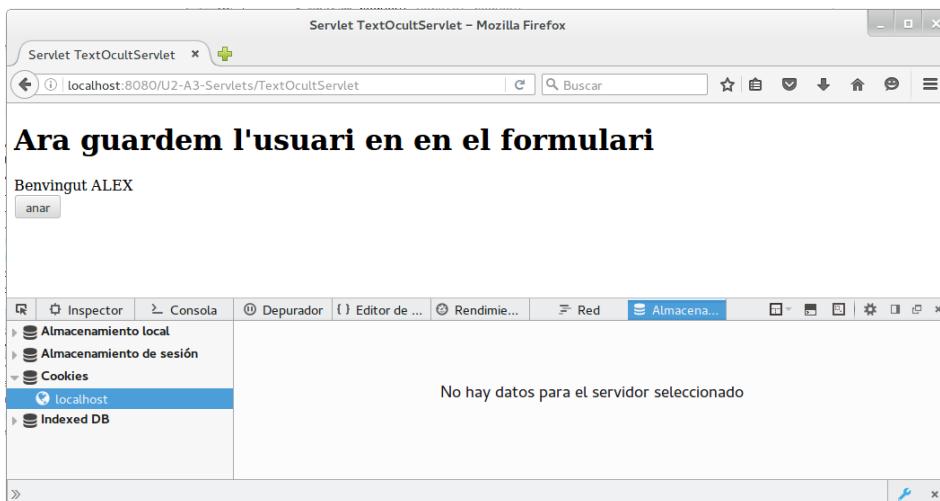
```
1 String n = request.getParameter("userName");
```

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.

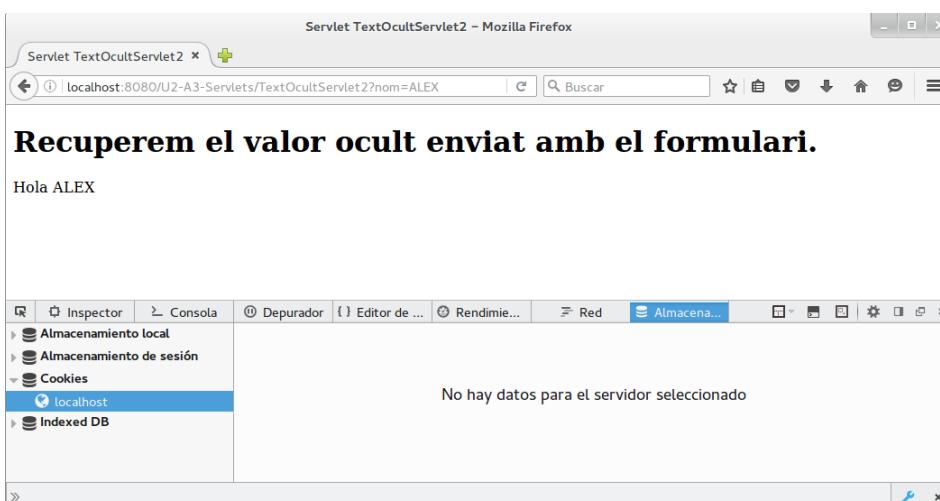
```

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 reescrituraURL.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="ReescrituraURLServlet" 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:

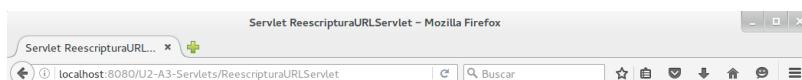
```
1 String n = request.getParameter("userName");
```

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.

```
1 String n = request.getParameter("userName");
2 out.print("<br><a href='ReescrituraURLServlet2?nom=" + n + "'>anar</a>");
```

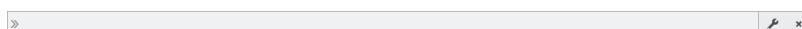
L'usuari només veurà un enllaç que li permet anar a una altra pàgina web del programa (vegeu la figura 3.7). El destí d'aquest enllaç serà un altre *servlet*, anomenat ReescrituraURLServlet2, que rebrà el paràmetre.

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



Ara cream un enllaç amb el nom com a paràmetre

Benvingut ALEX
[anar](#)



Finalment, només ens falta crear el *servlet* ReescrituraURLServlet2 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.

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

A la figura 3.8 podeu veure la pàgina que envia el *servlet* ReescrituraURLServlet2.

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 mostrerà 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:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Sessió</title>
5          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6      </head>
7      <body>
8          <h1>Emmagatzemar l'usuari a la sessió.</h1>
9          <h2>Introdueix el teu nom:</h2>
10         <form action="SessioServlet" method="post">
11             <label for="nom"> Nom:</label>
12             <input id="nom" type="text" name="userName"/> <br/>
13             <input type="submit" value="enviar"/>
14         </form>
15     </body>
16 </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>");
```

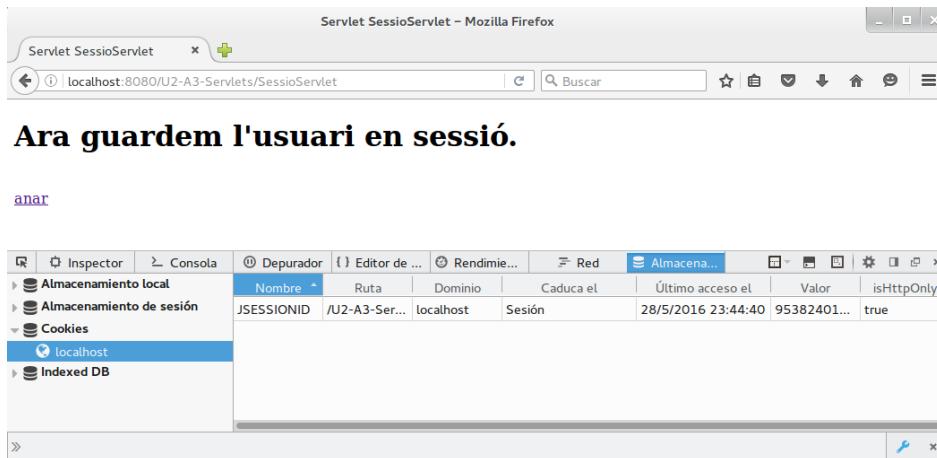
Primer de tot, obtenuï la sessió mitjançant la funció getSession() de l'objecte Request. Aquesta funció crea una nova sessió si no existeix una sessió creada prèviament. Si ja hi ha una sessió creada, s'utilitzarà la mateixa per poder manipular, si es vol, les dades que contingui.

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

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.

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.

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.

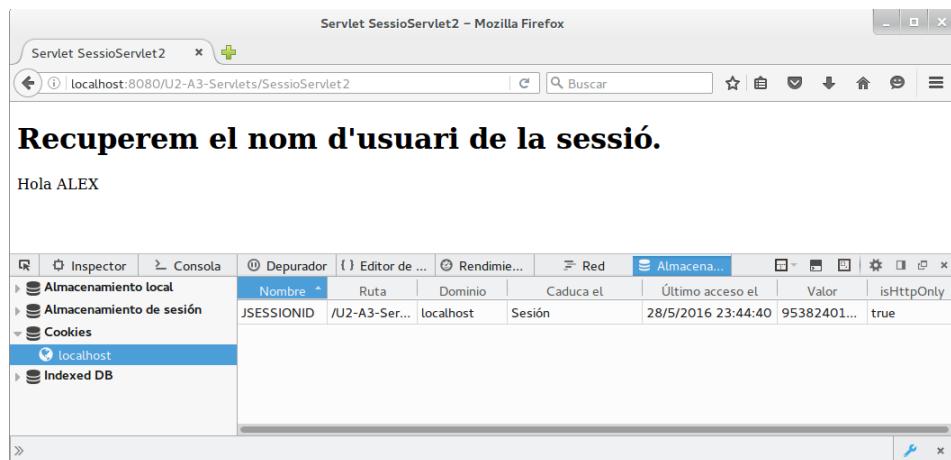
```
1 out.print("<br><a href='SessioServlet2'>anar</a>");
```

Modifarem la funció *processRequest* del *servlet SessioServlet2* 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 obtingut 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

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.

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.

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 compararà 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 {
```

```

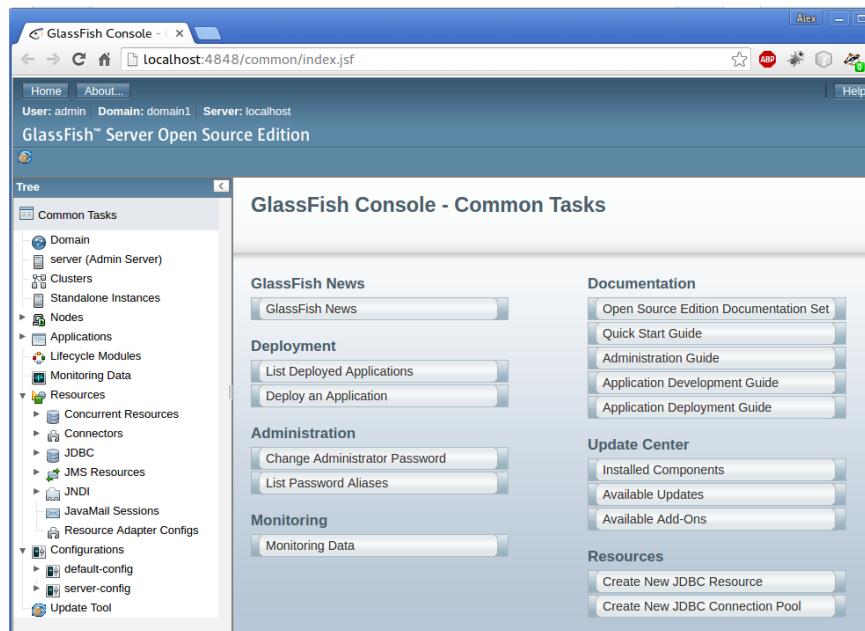
3
4     @Override
5     public String catalogar(String llibre){
6         return "Llibre " + llibre + " catalogat.";
7     }
8
9     @Override
10    public String veureDisponibilitat(String llibre){
11        if(llibre.equals("Java")){
12            return "Llibre " + llibre + " disponible.";
13        }
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

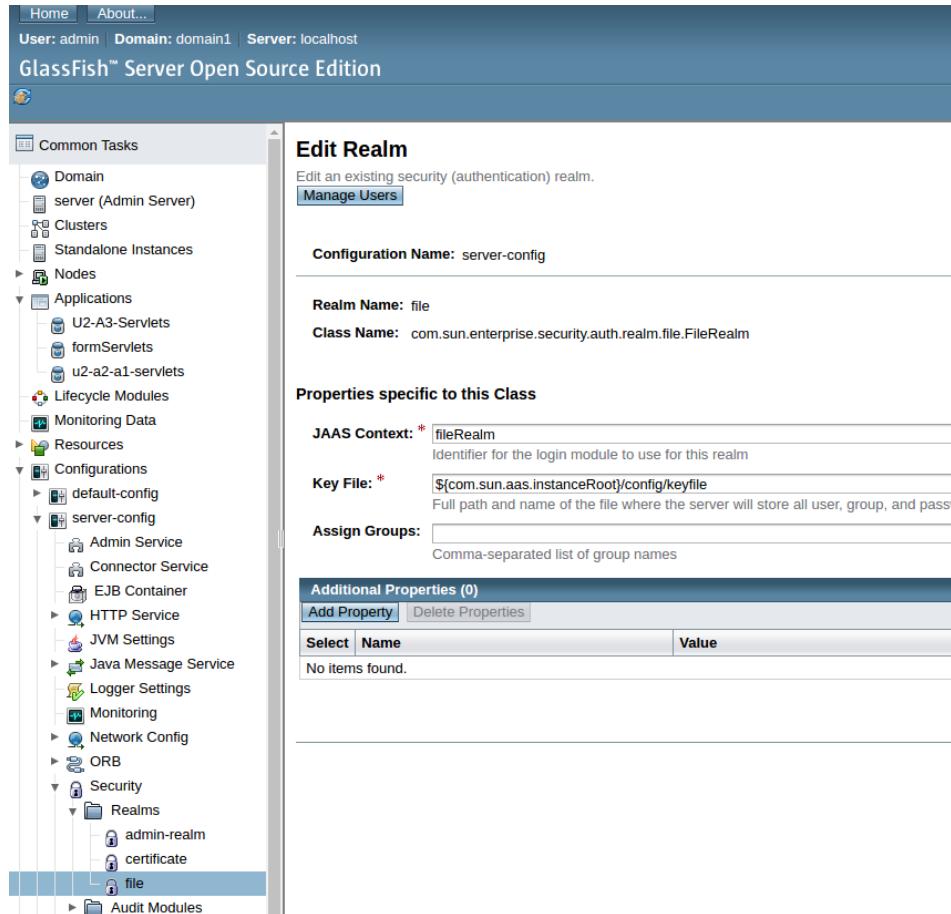


Per accedir a la consola podeu fer-ho posant a l'URL del navegador l'adreça localhost:4848/common/index.jsf, o bé amb el programa NetBeans accedint al

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 serveurs d'aplicacions ens proporcionen un mecanisme d'autenticació per a aplicacions petites.

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

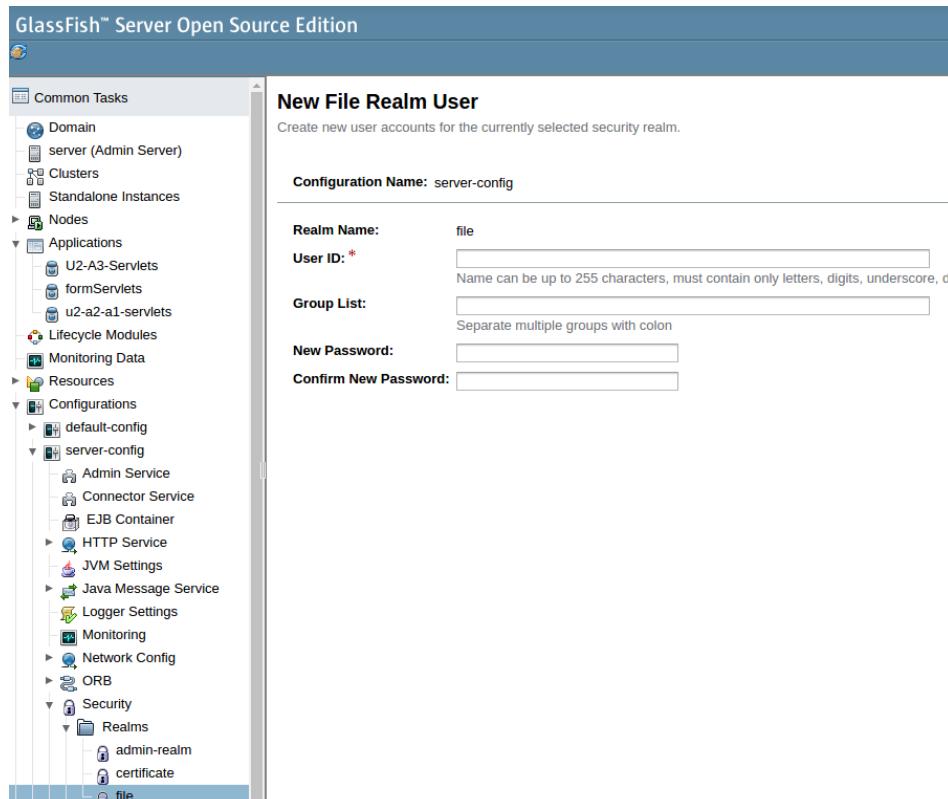
- **admin-realm:** és un fitxer on s'emmagatzemen les credencials dels usuaris administradors. Aquest fitxer s'anomena admin-keyfile.
- **certificate:** el servidor emmagatzema les credencials d'usuari en una base de dades de certificats. Quan s'utilitza aquesta opció, el servidor utilitzarà certificats 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

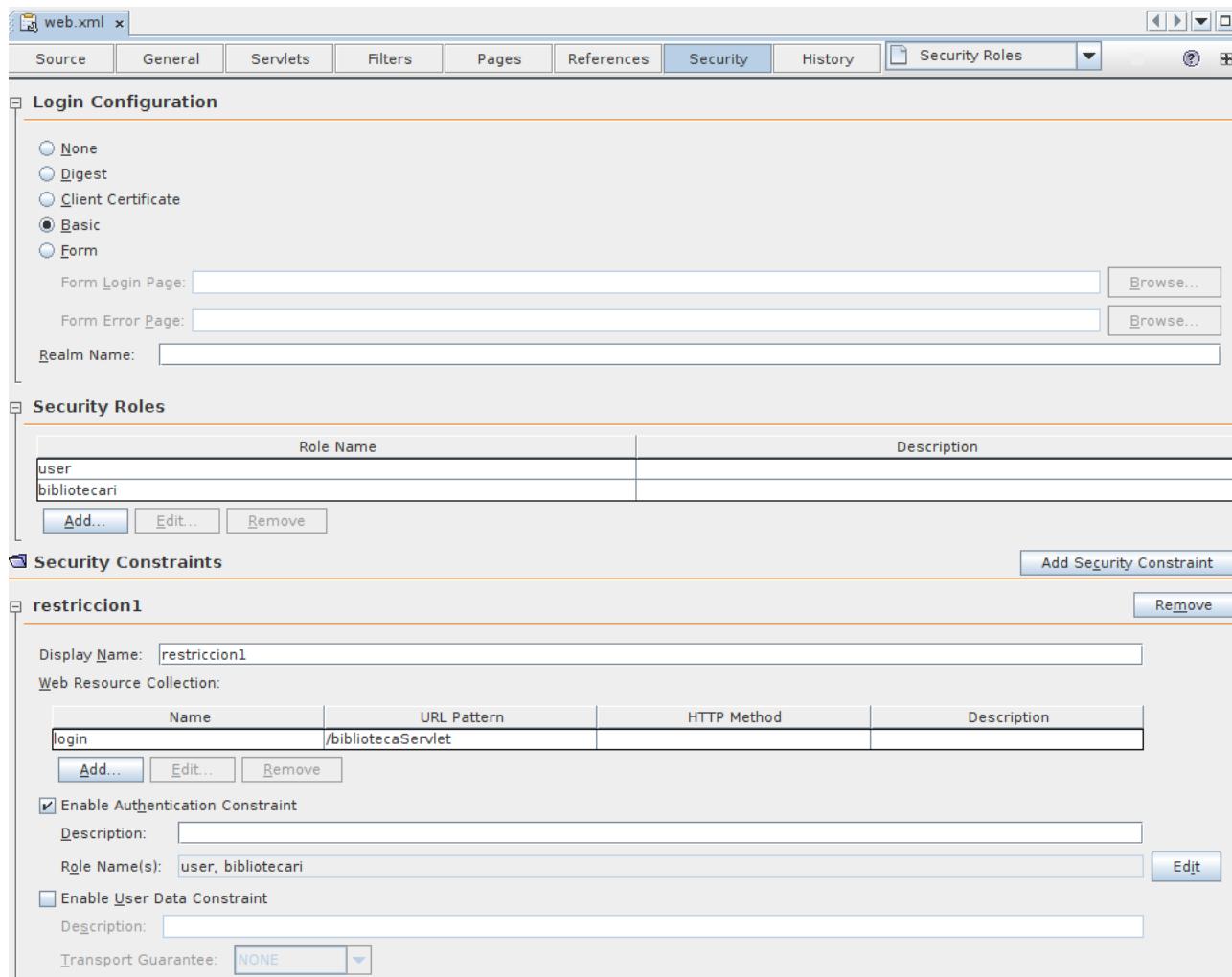
File Users (2)		
	New...	Delete
Select	User ID	Group List:
<input type="checkbox"/>	alex	bibliotecari
<input type="checkbox"/>	user	user

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.

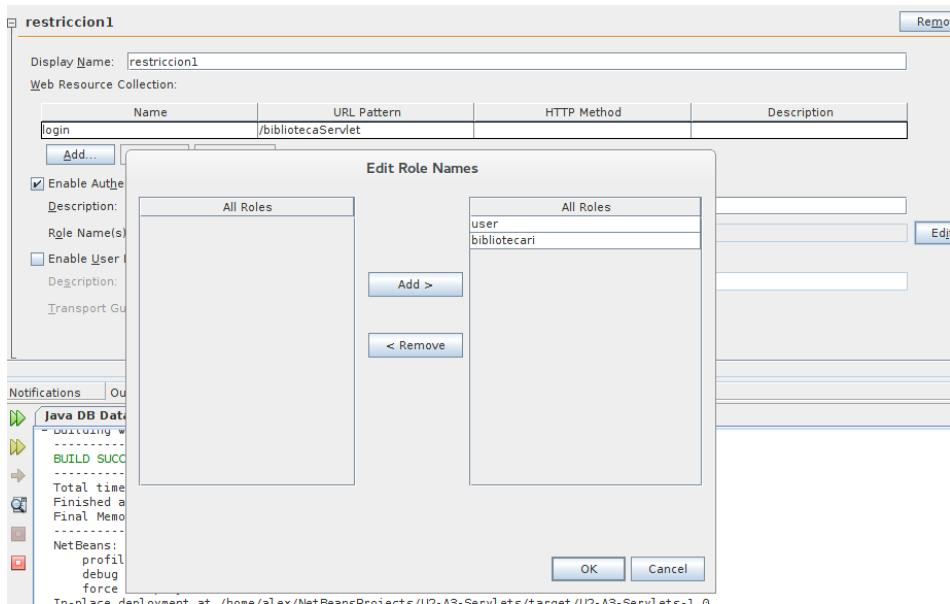
Aneu a l'apartat de seguretat del fitxer web.xml i afegiu-hi els rols, tal com podeu veure a la figura 3.15):

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'lsls 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 authentication 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:

```

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    <session-config>
4      <session-timeout>
5        30
6      </session-timeout>
7    </session-config>
8    <security-constraint>
9      <display-name>restriccion1</display-name>
10     <web-resource-collection>
11       <web-resource-name>login</web-resource-name>
12       <description/>
13       <url-pattern>/bibliotecaServlet</url-pattern>
14     </web-resource-collection>
15     <auth-constraint>
16       <description/>
17       <role-name>user</role-name>
18       <role-name>bibliotecari</role-name>
19     </auth-constraint>
20   </security-constraint>
21   <login-config>
22     <auth-method>BASIC</auth-method>
23   </login-config>
24   <security-role>
25     <description/>
26     <role-name>user</role-name>
27   </security-role>
28   <security-role>
29     <description/>
30     <role-name>bibliotecari</role-name>
31   </security-role>
32 </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

Default Principal To Role Mapping **Enabled**
 Apply default principal-to-role mapping at deployment when application-specific mapping is not defined; does not affect currently deployed applications

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 definitos 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, modifiqueu 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.";
16         }
17     }

```

```

16     }
17     return "Llibre " + llibre + " no disponible.";
18 }
19
20 @DenyAll
21 @Override
22 public Boolean demanarPrestec(String llibre){
23     return false;
24 }
25
26 }
```

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:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server
3   9.0 EJB 3.0//EN" "http://www.sun.com/software/appserver/dtds/sun-ejb-
4     jar_3_0-0.dtd">
5 <sun-ejb-jar>
6   <security-role-mapping>
7     <role-name>user</role-name>
8     <group-name>user-group</group-name>
9   </security-role-mapping>
10  <security-role-mapping>
11    <role-name>bibliotecari</role-name>
12    <group-name>bibliotecari-group</group-name>
13  </security-role-mapping>
14  <enterprise-beans/>
15 </sun-ejb-jar>
```

Ja heu acabat de donar l'autorització demandada 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.

```

1 @EJB
2 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*.

```

1 out.println("<p>Usuari: " + request.getUserPrincipal() + "</p>");
2 if(request.isUserInRole("bibliotecari")){
3     out.println("<p>" + b.catalogar("java") + "</p>");
4     out.println("<p>" + b.veureDisponibilitat("php") + "</p>");
5 }
```

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'accendir a `catalogar`, ja que no hi està autoritzat, però sí que podria accedir a `veureDisponibilitat`. El codi seria el següent:

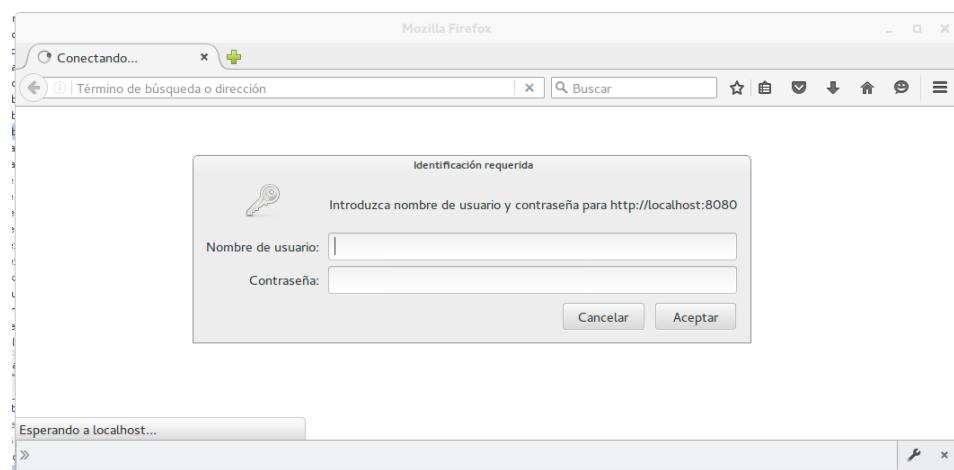
```
1 if(request.isUserInRole("user")){
2     out.println("<p>" + b.veureDisponibilitat("java") + "</p>");
3 }
```

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.

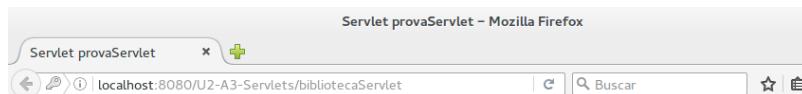
```
1 if(b.demanarPrestec("java")){
2     out.println("<p> Mai es veurà aquest text.</p>");
3 }
```

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'

Servlet provaServlet at /U2-A3-Servlets

Usuari: alex
 Llibre java catalogat.
 Llibre php no disponible.
 No tens els privilegis suficients per realitzar aquesta acció.

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'adreses 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.

Generació dinàmica de pàgines web

Mercedes Castellón Fuentes, Miguel Angel Lozano Márquez, Sergi Pérez Pérez

Desenvolupament web en entorn servidor

Índex

Introducció	5
Resultats d'aprenentatge	7
1 Introducció a Spring i Spring MVC	9
1.1 "Hola, Món" amb Maven	9
1.1.1 Preparació de l'entorn de desenvolupament	10
1.1.2 Actualització dels repositoris amb Maven	10
1.1.3 Creació d'un projecte amb Maven	11
1.2 'Hola, Món' web amb Maven	15
1.2.1 Creació d'un projecte nou	15
1.2.2 Estructura típica d'una aplicació web	17
1.3 "Hola, Món" amb Spring MVC	18
1.3.1 Creació del projecte "Hola, Món" a partir de Maven WebApp	18
1.3.2 Afegint dependències J2EE i Spring MVC	19
1.3.3 Arquitectura i procés bàsic d'una aplicació Spring MVC	20
1.3.4 Completant 'Hola, Món' amb l'arquitectura Spring MVC	22
1.3.5 Conclusions del nostre 'Hola, Món' Spring MVC	25
1.4 Estoc de medicaments, benvinguda	26
1.4.1 Creació i configuració inicial del projecte Estoc de medicaments	26
1.4.2 Pàgina de benvinguda	27
1.5 Què s'ha après?	29
2 Spring MVC, aplicació web	31
2.1 Estoc de medicaments, capa de domini	32
2.1.1 Creació del domini	33
2.1.2 Fent servir el domini	34
2.2 Estoc de medicaments, capa de persistència	37
2.2.1 Creació de la persistència	38
2.3 Estoc de medicaments, capa de servei	41
2.3.1 Adaptació del repositori	42
2.3.2 Creació del servei	43
2.3.3 Afegits a la capa de presentació	44
2.3.4 Què heu fet i què heu après amb la capa de servei?	46
2.4 Què s'ha après?	46
3 Spring MVC, aprofundint en els controladors	49
3.1 Estoc de medicaments, capa de servei a medicament	50
3.1.1 Creant el servei a medicament	50
3.1.2 Ordenant les crides des del controlador	51
3.2 Estoc de medicaments, medicaments per malaltia	52
3.2.1 Adaptant repositori i servei	53
3.2.2 Adaptant el controlador	53
3.3 "Estoc de medicaments", llista de medicaments segons filtre	55

3.3.1	Adaptant repositori i servei	55
3.3.2	Adaptant el controlador	57
3.4	"Estoc de medicaments", detall de medicament	58
3.4.1	Mostrant el detall	59
3.4.2	Arribant al detall des de la llista	61
3.5	Què s'ha après?	63
4	Spring MVC, altres aplicacions	65
4.1	Estoc de medicaments, formulari per afegir medicaments	66
4.1.1	Treballant les capes de domini, persistència i servei	67
4.1.2	Formulari a la capa de presentació	67
4.2	Estoc de medicaments, llista blanca al formulari	72
4.3	Estoc de medicaments, pàgina de 'login'	73
4.3.1	Configurant Spring Security	73
4.3.2	El controlador per al 'login'	76
4.3.3	La vistes	76
4.3.4	Provant d'entrar	78
4.4	Estoc de medicaments, internacionalització	79
4.4.1	Externalitzem els missatges	80
4.4.2	Configurant i implementant el canvi d'idioma	82
4.4.3	Provant a canviar d'idioma	82
4.5	Què s'ha après?	83

Introducció

El desenvolupament web en la part del servidor es pot fer amb diversos llenguatges i servidors d'aplicacions que els puguin executar. Cadascun d'aquests té els seus avantatges; per exemple, amb PHP la corba d'aprenentatge és ràpida, i amb JEE (Java Enterprise Edition) es poden desenvolupar aplicacions molt escalables i segures.

En aquesta unitat anirem una mica més enllà dels avantatges propis de JEE per afegir més productivitat en el desenvolupament i permetre un manteniment eficient i reusabilitat del codi molt gran. Això ho farem emprant el **framework Spring MVC** basat en Java Enterprise Edition, desenvolupant uns projectes preliminars seguits d'una veritable aplicació en aquest entorn.

Per introduir el *framework* explicarem les típiques aplicacions “Hola, Món” des de diversos punts de vista. En primer lloc, crearem projectes Maven amb l’estructura de qualsevol aplicació web fins a arribar a crear un “Hola, Món” amb l’estructura d’una aplicació web basada en Spring MVC.

Un cop feta la introducció, es començarà a desenvolupar tota una aplicació web que anomenarem **“Estoc de medicaments”**. L’anirem completant pas a pas explicant els continguts que hi afegim, és a dir, especificant els conceptes i la pràctica amb Spring MVC.

Farem una pàgina de benvinguda de l’aplicació que ens servirà per compondre l’estructura bàsica d’una aplicació web Spring MVC i descriure com una petició va des del navegador a un controlador frontal (*Dispatcher Servlet*) i aquest el dirigeix a un controlador específic.

Continuarem mostrant la llista de medicaments però desenvolupant totes les capes que Spring MVC recomana:

- presentació (*Presentation*)
- domini (*Domain*)
- serveis (*Services*)
- persistència (*Persistence*)

Aprofundirem en els controladors desenvolupant un moviment d'estoc en “Estoc de medicaments” i a partir de mostrar el detall. Veurem com podem passar paràmetres amb l’URL i amb el cos de la petició.

Finalment, veurem altres aplicacions del *framework*, com el tractament de formularis, l’autenticació i certs aspectes sobre la seguretat i la internacionalització.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Desenvolupa aplicacions web identificant i aplicant mecanismes per separar el codi de presentació de la lògica de negoci.

- Identifica els avantatges de separar la lògica de negoci dels aspectes de presentació de l'aplicació.
- Analitza tecnologies i mecanismes que permeten fer aquesta separació i les seves característiques principals.
- Utilitza objectes i controls en el servidor per generar l'aspecte visual de l'aplicació web en el client.
- Empra formularis generats de manera dinàmica per respondre als esdeveniments de l'aplicació web.
- Identifica i aplica els paràmetres relatius a la configuració de l'aplicació web.
- Escriu aplicacions web amb manteniment d'estat i separació de la lògica de negoci.
- Aplica els principis de la programació orientada a objectes.
- Aprova i documenta el codi.

1. Introducció a Spring i Spring MVC

Spring és un *framework* amb mòduls basat en Java Enterprise Edition. La principal característica del seu *core* és la utilització del patró de disseny inversió de control (IoC, per les sigles en anglès) i també la injecció de dependències (DI, per les sigles en anglès), un tipus d'IoC.

En concret, fent servir IoC i DI al nostre codi no crearem els objectes que necessitem de les llibreries del *framework*; simplement definint la configuració amb fitxers XML o amb anotacions al codi, Spring ens proporcionarà l'objecte adient.

Spring MVC és un mòdul d'Spring que ens ajuda a construir aplicacions sota el patró model vista controlador (MVC, per les sigles en anglès)

El model és el conjunt d'objectes que representen les dades de la nostra aplicació, la vista correspon a la manera de presentar les dades a l'usuari, i controlador manega les peticions fetes per l'usuari, interactuant amb la vista i amb el model.

A Spring MVC, qualsevol petició de l'usuari és recollida per un controlador central (*Dispatcher Servlet*) que determinarà, segons la configuració, el controlador específic de la nostra aplicació que haurà de recollir la petició. El nostre controlador haurà de construir la resposta amb la vista que correspon a la petició amb el model adient. El *Dispatcher Servlet* tornarà a prendre el control i retornarà la resposta al client.

I per fer independent els nostres projectes de l'IDE farem servir Maven com a eina que dóna estructura als projectes i que permet baixar les llibreries que fareu servir des de repositoris centrals a partir de la configuració de dependències.

Començareu creant dos projectes amb Maven sense dependències d'Spring MVC per conèixer l'estruatura dels projectes i les dependències necessàries amb Java Enterprise Edition per continuar amb projectes Spring MVC, un “Hola, Món” amb Spring MVC per apropar-nos al *framework* i un projecte “Estoc de medicaments” que simplement mostra una pàgina de benvinguda.

1.1 "Hola, Món" amb Maven

Maven és una eina que us permet descarregar totes aquelles llibreries que necessita un servidor extern el qual conté un repositori amb un llistat molt ampli de llibreries i components; a més de poder descarregar llibreries, també us permet la generació de projectes de manera automàtica per a aplicacions de propòsit específic. Per exemple, una aplicació web, com seria el vostre cas.

Maven es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Tiene un modelo de configuración basado en un formato XML (archivo pom.xml). Actualmente es un proyecto perteneciente a la Apache Software Foundation.

Maven també permet transportar un projecte des d'un entorn de desenvolupament a un altre. Per exemple, si teniu un projecte Maven creat amb Eclipse el podreu importar sense problemes a NetBeans.

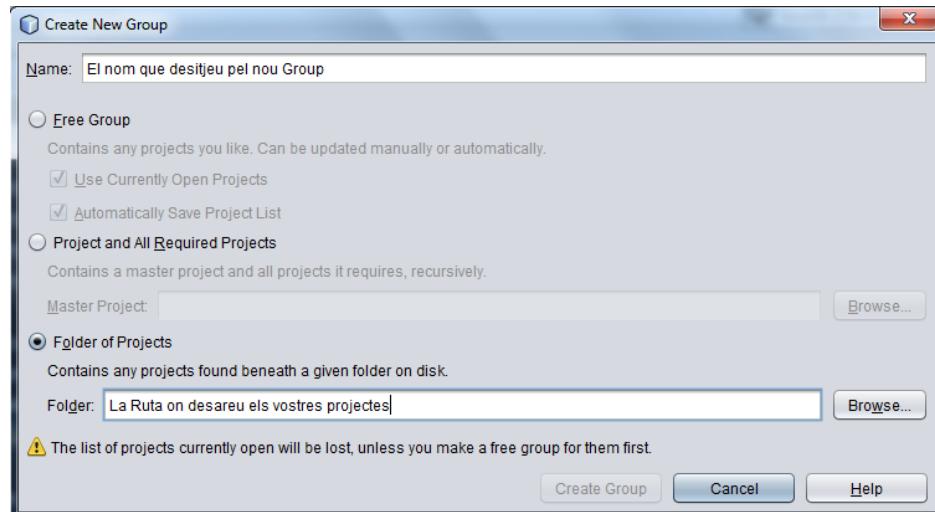
1.1.1 Preparació de l'entorn de desenvolupament

El primer que farem per preparar el vostre entorn serà configurar Netbeans per aíllar els vostres projectes dels projectes d'altres mòduls desenvolupats amb Netbeans, i d'aquesta manera aconseguireu un espai de treball molt més net.

Per tant, aneu al directori *NetBeansProjects* i allà creeu el directori *Spring_Projects*, que serà el que fareu servir per als vostres projectes amb Spring.

Un cop creat el directori, torneu a Netbeans i aneu a *File/Projects Group*, això us obrirà una finestra que us permetrà definir el directori que acabeu de crear com un nou espai de treball en blanc (vegeu la figura 1.1).

FIGURA 1.1. Creació d'un grup de projectes



1.1.2 Actualització dels repositoris amb Maven

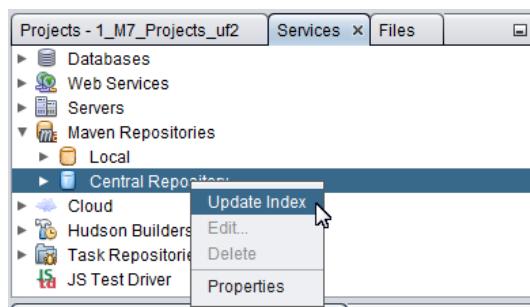
Un cop definit i seleccionat el nou espai de treball, el que fareu ara serà actualitzar els repositoris amb Maven.

Per això, en el vostre entorn de Netbeans aneu a la pestanya *Serveis* i allà desplegueu l'apartat *Maven Repositories*; com podeu observar, existeixen dos repositoris: un d'extern, o repositori central, i un local.

El que fareu serà establir una connexió amb el repositori central per tal d'obtenir una llista actualitzada de tots els components emmagatzemats al repositori central.

Per fer-ho, aneu a la pestanya *Services* de NetBeans, desplegueu l'element *Maven Repositories* i, situats al node *Central Repository*, prement amb el botó dret, podreu actualitzar la llista (opció *Update Index*), tal com es pot veure en la figura 1.2.

FIGURA 1.2. Actualització de l'índex de components

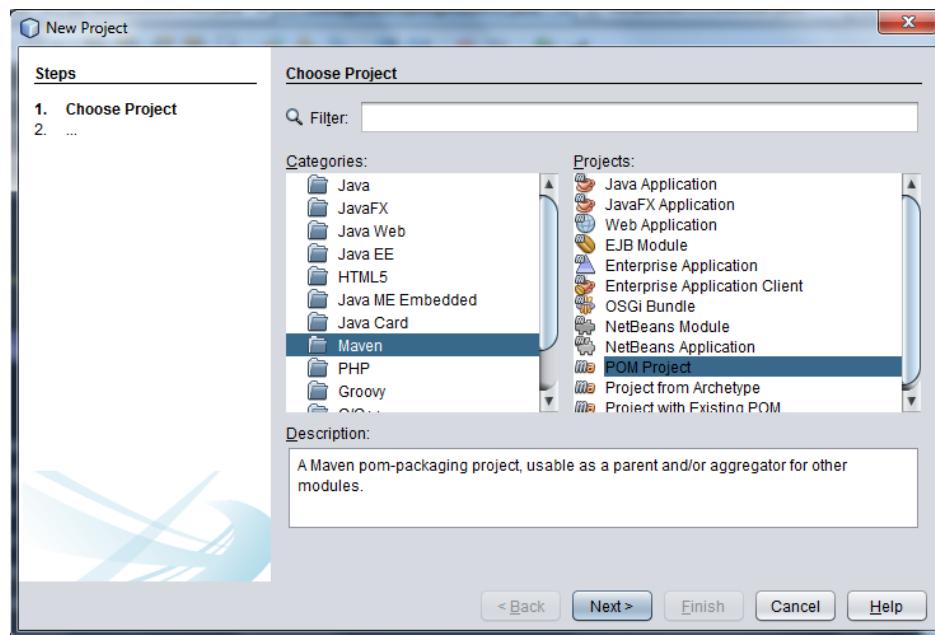


El procés d'actualització dels repositoris triga una bona estona; per tant, tingueu paciència.

1.1.3 Creació d'un projecte amb Maven

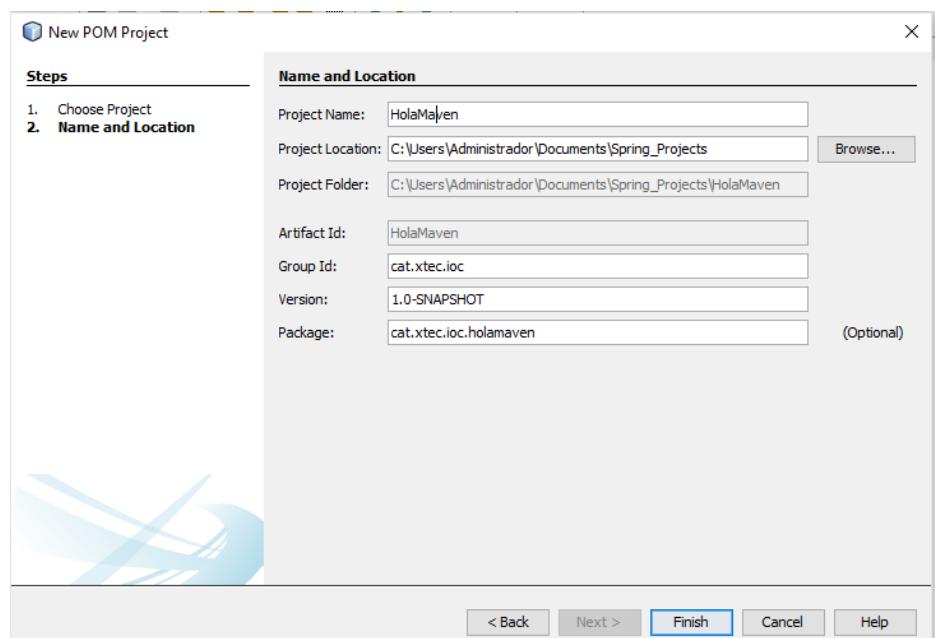
Un cop actualitzats els repositoris podeu procedir a la creació d'un projecte amb Maven; a partir d'aquest aprofundirem en els conceptes i l'estructura que en proporciona.

Per fer això aneu a *File/New Project*, en l'apartat de categories seleccioneu *Maven* i en l'apartat *Project* seleccioneu *POM Projects*. Ara només us resta definir el nom i la localització del vostre primer projecte amb Maven (vegeu la figura 1.3).

FIGURA 1.3. Nou projecte Maven

Aquí haureu d'especificar tres apartats: el nom del projecte, la localització del projecte i l'identificador del grup (vegeu la figura 1.4).

- nom del projecte: HolaMaven
- localització del projecte: Spring_Projects
- identificador del grup: cat.xtec.ioc

FIGURA 1.4. Propietats del projecte

Si tot ha anat bé, tant al directori *Spring_Projects* com a Netbeans podreu veure que s'ha creat un nou projecte.

Definició de les dependències

Les dependències no són altra cosa que les llibreries o els components que necessita la nostra aplicació per funcionar o portar a terme una tasca determinada. Un exemple de dependència seria la llibreria JDBC per a MySQL, que ens permet obrir i tancar connexions contra una base de dades MySQL i fer diferents operacions contra aquesta.

Per desenvolupar una aplicació web necessiteu descarregar tres dependències des dels repositoris de Maven, que són:

- l'entorn de treball Spring
- la llibreria jstl
- la llibreria servlet-api

Maven fa servir un fitxer de configuració XML anomenat pom.xml on podeu indicar les dependències de la vostra aplicació web; aquest fitxer el podeu localitzar a la carpeta *Projects Files* del vostre nou projecte. Però com podeu veure si l'obriu, no fa cap esment de Spring ni a la resta de llibreries que comentàvem.

L'edició del fitxer pom.xml per incloure aquestes tres llibreries la podem portar a terme de dues maneres: una seria fent-la directament contra el fitxer pom.xml i l'altra seria utilitzant l'assistent que ens ofereix NetBeans per tal de fer aquesta edició contra el fitxer pom.xml.

En el vostre cas, i per començar, fareu servir l'assistent, ja que us ajudarà a comprendre d'una manera més senzilla què esteu fent.

Així doncs, amb el fitxer pom.xml obert, aneu a la carpeta *Dependències* del vostre projecte i feu clic amb el botó dret del ratolí. Això obrirà un desplegable amb l'opció *Add Dependèncie...*; feu clic sobre aquesta opció i s'obrirà una finestra nova.

En aquesta finestra apareixen diferents camps per omplir, però a nosaltres només ens interessen tres, que són:

- *Group Id*: es correspon amb l'identificador que fa servir Maven per a un conjunt de components desenvolupats per un determinat projecte o empresa. Aquest identificador pren l'aspecte de l'espai de nom del projecte, com per exemple *org.springframework*.
- *Artifact Id*: es correspon amb un component determinat desenvolupat per un projecte o empresa; en general, és el nom del component o llibreria.
- *Version*: representa la versió del component que volem fer servir.

Ompliu els tres camps que hem comentat amb els següents valors, respectivament:

- org.springframework

- spring-webmvc
- 4.0.3.RELEASE

Afegiu la dependència i observeu què ha passat amb el fitxer pom.xml i amb la carpeta *Dependències*.

Com podeu veure, el fitxer pom.xml s'ha modificat amb noves etiquetes i valors, i la carpeta *Dependències* incorpora ara noves llibreries que es corresponen amb les llibreries que ens permeten desenvolupar aplicacions web amb l'entorn de treball Spring.

Si haguéssiu tingut un error d'escriptura dels valors indicats per a la dependència, Maven seria incapàc de descarregar-la i ho indicaria amb un petit triangle groc d'avertència sobre la icona del component a la carpeta *Dependències*.

Un cop descarregat el primer component, descarregueu els altres dos amb l'assistent. Les dades que necessiteu són les següents:

- Per a la llibreria jstl
 - javax.servlet
 - jstl
 - 1.2
- Per a la llibreria servlet-api
 - javax.servlet
 - javax.servlet-api
 - 3.1

El fitxer pom.xml del projecte quedarà de la següent manera:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5                           http://maven.apache.org/xsd/maven-4.0.0.xsd
6           ">
7     <modelVersion>4.0.0</modelVersion>
8     <groupId>cat.xtec.ioc</groupId>
9     <artifactId>A0010_Spring_Introduccio</artifactId>
10    <version>1.0-SNAPSHOT</version>
11    <packaging>pom</packaging>
12    <dependencies>
13      <dependency>
14        <groupId>org.springframework</groupId>
15        <artifactId>spring-webmvc</artifactId>
16        <version>4.0.3.RELEASE</version>
17      </dependency>
18      <dependency>
19        <groupId>javax.servlet</groupId>
20        <artifactId>jstl</artifactId>
21        <version>1.2</version>
22      </dependency>
23      <dependency>

```

```

24      <artifactId>javax.servlet-api</artifactId>
25      <version>3.1.0</version>
26    </dependency>
27  </dependencies>
28  <properties>
29    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
30  </properties>
31</project>

```

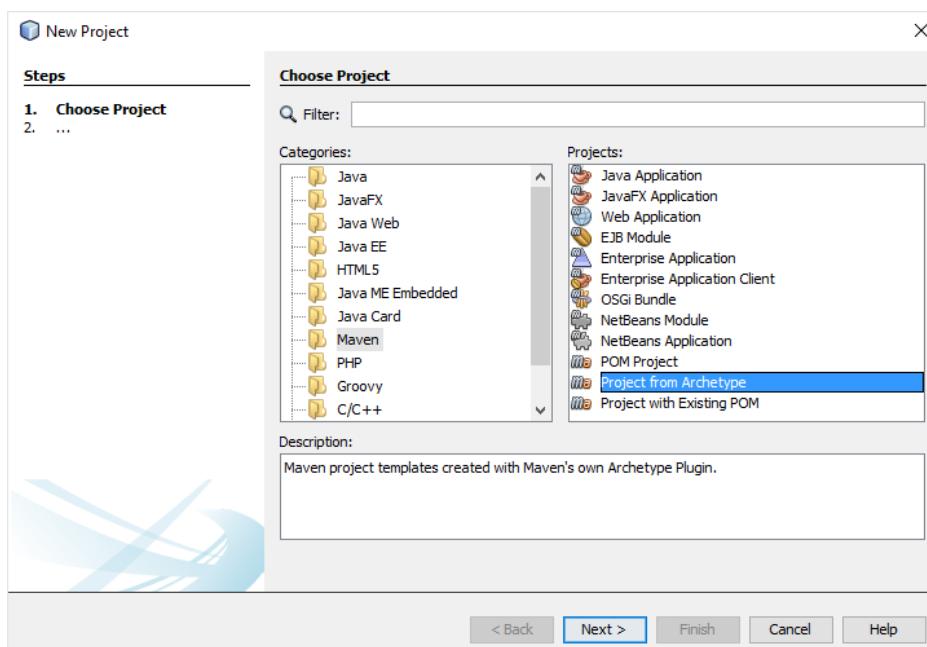
1.2 'Hola, Món' web amb Maven

La creació d'aquest segon projecte la farem mitjançant un arquetipus, que és la definició d'una estructura de directoris i fitxers específics per a un projecte genèric, i el que fa és permetre la creació d'un projecte nou seguint una convenció determinada; en el vostre cas, es crearà una estructura que segueix la convenció establerta per a la creació d'aplicacions web amb Spring.

1.2.1 Creació d'un projecte nou

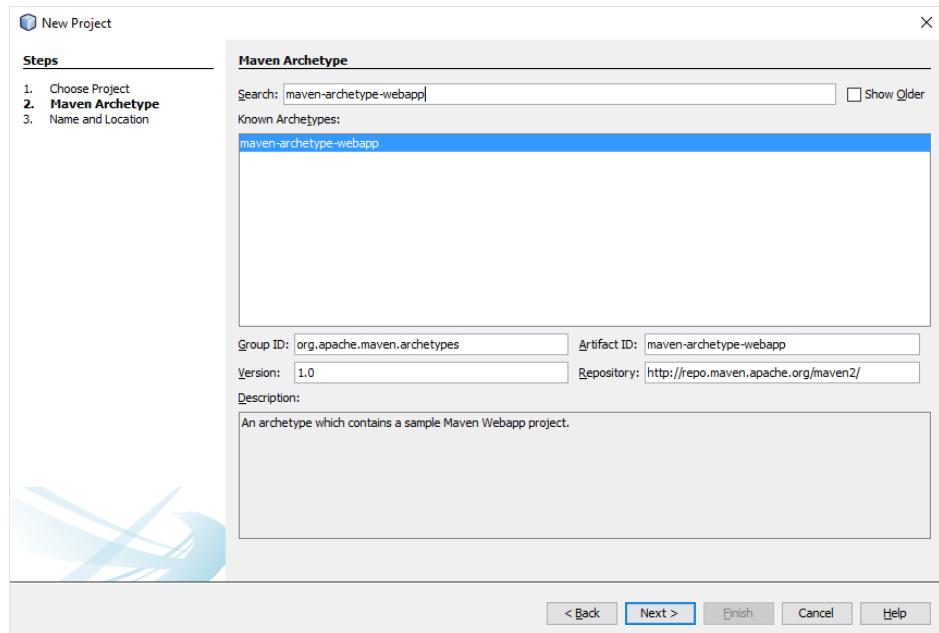
Per crear aquest nou projecte web aprofitarem la carpeta *Modules* del projecte Maven creat anteriorment. Si feu clic sobre la carpeta amb el botó dret de ratolí s'obrirà un menú contextual, on podreu seleccionar l'opció *Create New Module*. Això obrirà un nou assistent per a la creació de projectes. Aquí seleccionareu la categoria *Maven*, i dintre del tipus de projectes disponibles seleccionareu *Project from Archetype* (vegeu la figura 1.5).

FIGURA 1.5. Nou projecte Maven from Archetype



Un cop seleccionat el nou tipus d'aplicació a desenvolupar, cercareu l'arquetipus desitjat; en el vostre cas heu de cercar el següent arquetipus: maven-archetype-webapp. Un cop localitzat l'arquetipus, el seleccionareu i pulsareu el següent en l'assistent (vegeu la figura 1.6).

FIGURA 1.6. Propietats de l'arquetipus

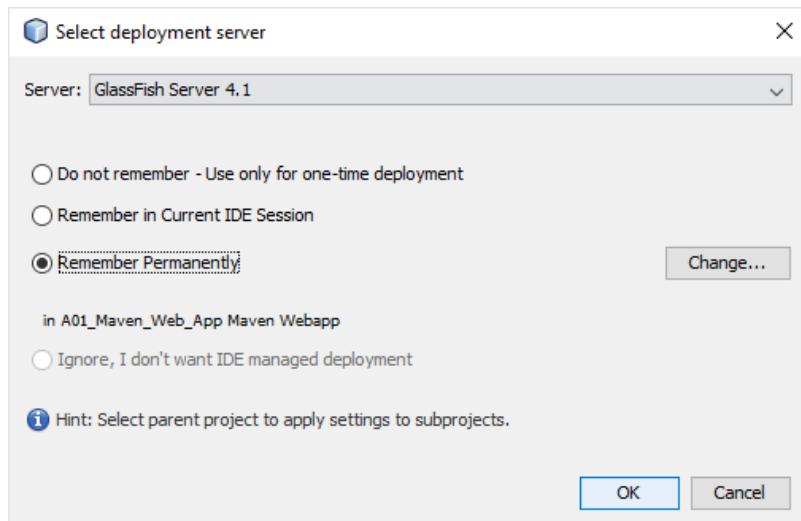


Això obrirà una nova finestra en l'assistent, on donareu el següent nom al projecte: A01_Maven_Web_App, i verificareu que tant la localització del projecte (Spring_Projects) com el Group Id (cat.xtec.ioc) són els correctes, i un cop fet això finalitzareu l'assistent.

En tancar l'assistent es generarà un nou mòdul, així com un nou projecte. Si el nom no es correspon amb el que li havíeu indicat prèviament podeu reanomenar-lo amb el botó dret de ratolí obrint el menú contextual i seleccionant l'opció *Reanomenar*. Aquest punt l'heu de fer des del projecte generat, no des del mòdul. Finalment, amb el mateix menú contextual teniu l'opció d'executar el nou projecte web clicant sobre l'apartat *Run*. En aquest punt s'obrirà un nou assistent on haureu d'indicar qui serà el servidor de desplegament de l'aplicació; seleccioneu *GlassFish Server 4.1* (o la versió més recent) i l'opció *Recorda de manera permanent* (vegeu la figura 1.7).

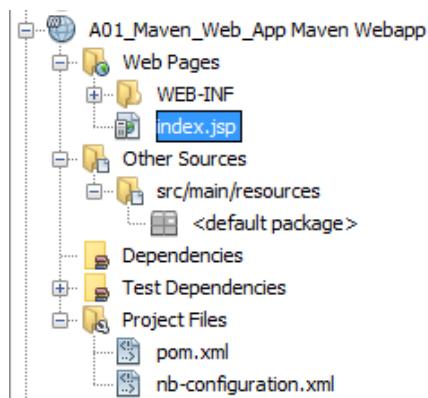
Després d'això, Netbeans procedirà al desplegament de la nostra aplicació, engregarà el servidor d'aplicacions GlassFish, executarà l'aplicació i podreu veure el resultat a través del vostre navegador web.

Si el desplegament s'ha fet correctament podeu anar a la pestanya *Services* de NetBeans i veure la vostre aplicació web com a node de *Servers/Glassfish Server xxx/Applications*, on “xxx” és la versió del servidor que heu fet servir.

FIGURA 1.7. Selecció del servidor d'aplicacions

1.2.2 Estructura típica d'una aplicació web

Si us situeu sobre el projecte A01_Maven_Web_App, la vostra primera aplicació té la següent estructura des de la pestanya projectes: *Web Pages*, *Other Sources*, *Dependencies*, *Test Dependencies* i *Project Files* (vegeu la figura 1.8).

FIGURA 1.8. Estructura d'un projecte Maven WebApp

La carpeta *Project Files* conté el fitxer pom.xml, que és on es declaren les dependències del projecte. Com podeu observar, el fitxer pom.xml del mòdul (projecte A01_Maven_Web_App) és diferent del fitxer pom.xml del projecte arrel (projecte A00_HolaMaven); tot i això, si obriu la carpeta *Dependencies* del projecte A01_Maven_Web_App podreu observar que fa referència a les dependències declarades en el projecte arrel.

La carpeta *Test Dependencies* inclourà les llibreries necessàries per fer les proves de la nostra aplicació. *Other Sources* és una carpeta que no farem servir per ara, i finalment *Web Pages* contindrà tot allò vinculat a les vistes web de l'aplicació. Dintre de *Web Pages* hi ha la carpeta *WEB-INF*, que serà l'arrel de totes les vistes generades.

Generalment, *WEB-INF*, a banda dels *scripts* jsp, contindrà el descriptor de desplegament web.xml i els fitxers de configuració de Spring que veurem més endavant.

1.3 "Hola, Món" amb Spring MVC

Fins ara heu après a crear projectes amb Maven, gestionar les dependències i crear una aplicació web molt senzilla però sense fer servir Spring.

Ara farem la nostra primera aplicació amb el *framework* Spring MVC amb l'objectiu de mostrar com es configura tot l'entorn d'un projecte desenvolupat amb aquest *framework*.

L'aplicació serà un “Hola, Món” per mostrar simplement aquest text en el navegador, però amb l'estructura d'un projecte Maven WebApp i els components de Spring MVC.

El projecte sencer de l'aplicació es pot baixar des del següent .

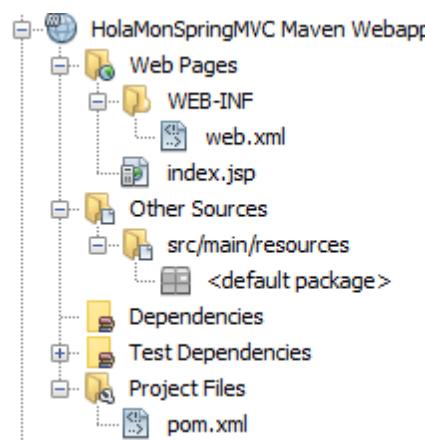
És convenient fer servir el projecte anterior per consultar, però és millor anar constraint el vostre propi projecte amb les indicacions que es van donant. No obstant això, si voleu executar el projecte sencer segurament us obligarà a fer *clean and build* abans.

1.3.1 Creació del projecte "Hola, Món" a partir de Maven WebApp

Creeu un nou projecte *Maven/Project from Archetype* amb l'arquetipus *maven-archetype-webapp* i anomeneu-lo HolaMonSpringMVC.

Recordem l'estructura del projecte creat fins ara: només és un projecte Maven WebApp a tots els efectes (vegeu la figura 1.9).

FIGURA 1.9. Estructura Maven d'una aplicació Spring MVC



Aquesta estructura correspon a qualsevol tipus d'aplicació, i per això, des de la pestanya *Files* de NetBeans o des del mateix explorador de fitxers del sistema operatiu creeu la carpeta *Java* dins de la carpeta *src/main*. Ara, si tornem a la pestanya *Projects* de NetBeans, veurem que ha aparegut la carpeta *Source Packages* on crear els nostres paquets.

Abans de continuar, canvieu el fitxer de configuració *web.xml* a una versió més moderna de JAVA EE.

```

1 <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
4                         http://java.sun.com/xml/ns/javaee/web-app_3_0.
5                         xsd">
6     <display-name>HolaMonSpringMVC</display-name>
7 </web-app>
```

Ara ja podeu executar l'aplicació seleccionant el servidor Glassfish i fent que recordi aquesta decisió permanentment (vegeu la figura 1.10).

FIGURA 1.10. Sortida de l'aplicació “Hola, Món” Spring MVC



1.3.2 Afegint dependències J2EE i Spring MVC

Anem a afegir les dependències que faran que el servidor d'aplicacions Glassfish, en executar la nostra aplicació, pugui crear els objectes necessaris de J2EE i Spring MVC i així aprofitar la funcionalitat que aquests entorns ens ofereixen.

A la secció *dependències* de pom.xml heu d'afegir-hi les següents:

```

1 <dependency>
2     <groupId>org.springframework</groupId>
3     <artifactId>spring-webmvc</artifactId>
4     <version>4.0.3.RELEASE</version>
5   </dependency>
6   <dependency>
7     <groupId>javax.servlet</groupId>
8     <artifactId>jstl</artifactId>
9     <version>1.2</version>
10  </dependency>
11  <dependency>
12    <groupId>javax.servlet</groupId>
13    <artifactId>javax.servlet-api</artifactId>
14    <version>3.1.0</version>
15  </dependency>
16  <dependency>
17    <groupId>javax</groupId>
18    <artifactId>javaee-web-api</artifactId>
19    <version>7.0</version>
20  </dependency>
```

A més, aprofitareu per fer servir la codificació UTF8 en tot el projecte afegint la propietat que defineix aquesta característica en el mateix fitxer pom.xml:

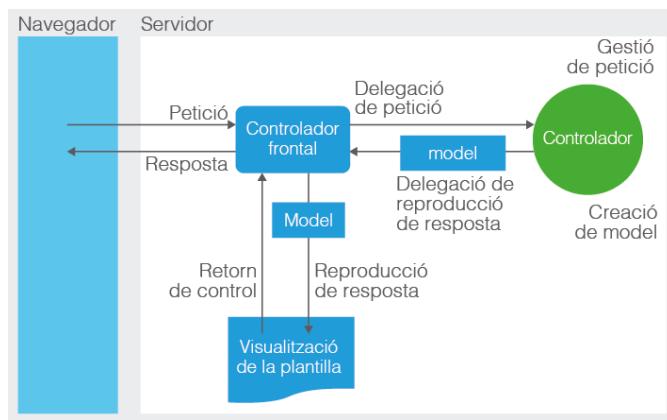
```

1 <properties>
2   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3 </properties>
```

1.3.3 Arquitectura i procés bàsic d'una aplicació Spring MVC

Des que un navegador fa una petició al servidor d'aplicacions, per exemple Glassfish, fins que el servidor construeix la resposta i retorna al navegador, el flux de l'aplicació és el que es mostra en la figura 1.11.

FIGURA 1.11. Arquitectura i procés d'una petició amb Spring MVC



A continuació es descriu la seqüència del procés petició-resposta.

1. El navegador demana (petició) un recurs a l'aplicació web.
2. El Front Controller o Dispatcher Servlet, implementat com un *dispatcher*, intercepta la petició i la delega al controlador adient, que ha de gestionar (*handler*) aquesta petició. Com veurem més endavant, la determinació del controlador adient es fa mitjançant la configuració dels *Handler Mappings*.
3. El controlador processa la petició i retorna al Front Controller el model i la vista. Aquest retorn es fa amb un objecte del tipus *ModelAndView*.
4. El Front Controller resol la vista actual (per exemple, un jsp) consultant l'objecte *ViewResolver*.
5. La vista seleccionada és retornada com a resposta al navegador que va fer la petició.

Front Controller

Front Controller és un patró de disseny que proposa la centralització de la gestió de les peticions i respostes. Cerqueu "front controller java enterprise edition" i trobareu documentació d'Oracle sobre aquest tema.

Web Application Context

Desplegar una aplicació Spring al servidor d'aplicacions, per exemple a Glassfish, és com si l'aplicació s'estigués executant en segon pla dins del servidor.

Els objectes de la nostra aplicació desplegada resideixen en **contenidors**.

Per a tota l'aplicació hi ha un contenidor arrel anomenat *Application Context* que conté els objectes definits al fitxer de configuració anomenat applicationContext.xml.

El nom del fitxer de configuració per a *Application Context* es pot canviar a web.xml, com mostra l'exemple següent.

```

1 <context-param>
2   <param-name>contextConfigLocation</param-name>
3   <param-value>/WEB-INF/rootApplicationContext.xml</param-value>
4 </context-param>
```

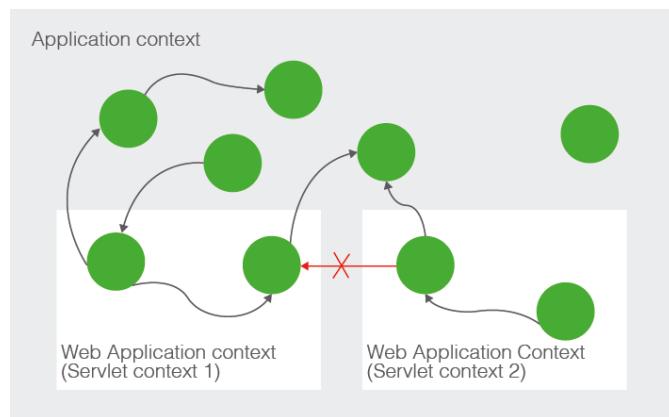
Per a cada Dispatcher Servlet es crearà un contenidor específic anomenat *Web Application Context*. Per defecte, el seu fitxer de configuració s'anomena xxx-servlet.xml, on “xxx” és el nom del Dispatcher Servlet.

El nom del fitxer per a *Web Application Context* es configura al mateix web.xml. L'exemple següent mostra com es configura un Dispatcher Servlet amb nom Dispatcher Servlet i amb fitxer de configuració /WEB-INF/spring/DispatcherServlet-servlet.xml.

```

1 <servlet>
2   <servlet-name>DispatcherServlet</servlet-name>
3   <servlet-class>org.springframework.web.servlet.DispatcherServlet</
4     servlet-class>
5   <init-param>
6     <param-name>contextConfigLocation</param-name>
7     <param-value>/WEB-INF/spring/DispatcherServlet-servlet.xml</param-
8       value>
9   </init-param>
10  <load-on-startup>1</load-on-startup>
11 </servlet>
```

Com es veurà més endavant, els fitxers de configuració de context, tant el de tota l'aplicació com els específics de cada Dispatcher Servlet, defineixen els objectes que es crearan al contenidor. La diferència és que els objectes a nivell d'aplicació es poden fer servir des de qualsevol contenidor, i els de nivell de Dispatcher Servlet, únicament des d'objectes del mateix contenidor. A més, els del contenidor a nivell d'aplicació poden fer servir qualsevol objecte de qualsevol contenidor específic, tal com es vol reflectir a la figura 1.12.

FIGURA 1.12. Visibilitat d'objectes segons els nivells

En aquests fitxers de configuració es defineixen quins objectes formaran part del contingidor, tant si són propis de l'aplicació com si són proporcionats per Spring o Java Enterprise Edition. Aquests objectes s'anomenen *Spring-managed beans* o simplement *beans*, i els farem servir al nostre codi declarant-los mitjançant anotacions, és a dir, no els instanciarem, perquè farem servir els objectes del contingidor. Per això es diu que l'objecte s'**injecta** quan es necessita i podem fer-lo servir (**injecció de dependències** o **inversió de control**)

Quan es dissenya una aplicació és important tenir en compte que els *beans* que defineixen la lògica de negoci, la interacció amb la persistència i altres interaccions s'han de compartir entre tots els *servlets*, i per això els definirem a nivell d'*ApplicationContext* (el contingidor general). En canvi, els controladors que maneguen peticions, els *View Resolvers* i altres com alguns que gestionen missatges, els ubicarem a nivell de *Web Application Context* (els específics de cada *Dispatcher Servlet*).

Injecció de dependències

La injecció de dependències o inversió de control aconsegueix codi més desacoblat, ens facilita els tests i, a més, ens permetrà canviar bocins de codi de manera més fiable i ràpida.

1.3.4 Completant 'Hola, Món' amb l'arquitectura Spring MVC

Un cop sabeu com ha de ser el flux d'una aplicació Spring MVC i els components que ha de tenir, desenvolupeu les classes i la configuració que falten a la nostra aplicació "Hola, Món".

En el vostre cas, deixareu que Spring agafi la configuració per defecte a nivell d'*ApplicationContext*. Per això no creareu cap fitxer de configuració de l'*ApplicationContext* (applicationContext.xml).

El que sí que hem de crear és la configuració del *Web Application Context*. Com només tindreu un *Dispatcher Servlet* amb el nom *Dispatcher Servlet*, podeu crear una nova carpeta de nom *spring* dins de *WEB-INF* i dins el fitxer DispatcherServlet.xml. El contingut és el que es mostra a continuació.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:context="http://www.springframework.org/schema/context"
5      xmlns:mvc="http://www.springframework.org/schema/mvc"
```

```

6      xsi:schemaLocation="http://www.springframework.org/schema/beans
7          http://www.springframework.org/schema/beans/spring-
8              beans.xsd
9          http://www.springframework.org/schema/context
10         http://www.springframework.org/schema/context/spring-
11             -context-4.0.xsd
12
13
14     <mvc:annotation-driven />
15     <context:component-scan base-package="cat.xtec.ioc" />
16
17     <bean class="org.springframework.web.servlet.view.
18         InternalResourceViewResolver">
19         <property name="prefix" value="/WEB-INF/views/" />
20         <property name="suffix" value=".jsp" />
21     </bean>
22
23 </beans>
```

Amb `<mvc:annotation-driven/>` li diem a Spring MVC que farem servir els *beans* DefaultAnnotationHandlerMapping, AnnotationMethodHandlerAdapter i ExceptionHandlerExceptionResolver. Aquests *beans* són necessaris per enviar (*dispatch*) les peticions als controladors que les maneguen.

Com s'ha dit a la part d'arquitectura i procés de Spring MVC, el Dispatcher Servlet ha d'identificar quin és el controlador que manegarà la petició. Per fer això cercarà tots els objectes amb l'anotació @Controller dels paquets indicats en la propietat base-package de l'etiqueta context:component-scan.

La darrera etiqueta *bean* indica a Spring la creació d'un objecte bean a partir de la classe InternalResourceViewResolver. Un View Resolver ajuda el Dispatcher Servlet a construir la resposta a partir de determinada vista. Spring MVC proporciona diverses implementacions de View Resolvers, i InternalResourceViewResolver és una d'aquestes.

Per definir el nostre Front Controller afegiu el següent contingut al fitxer de configuració web.xml.

```

1 <servlet>
2     <servlet-name>DispatcherServlet</servlet-name>
3     <servlet-class>org.springframework.web.servlet.DispatcherServlet</
4         servlet-class>
5     <init-param>
6         <param-name>contextConfigLocation</param-name>
7         <param-value>/WEB-INF/spring/DispatcherServlet-servlet.xml</param-
8             value>
9     </init-param>
10    <load-on-startup>1</load-on-startup>
11 </servlet>
12
13 <servlet-mapping>
14     <servlet-name>DispatcherServlet</servlet-name>
15     <url-pattern>/</url-pattern>
16 </servlet-mapping>
```

Fixeu-vos que es fa referència al fitxer de configuració DispatcherServlet-servlet.xml que hem creat. Podem canviar el nom del fitxer de configuració, però hauríeu de canviar-lo també aquí.

Continuant la compleció de la vostra aplicació seguint l'arquitectura i el model bàsic de Spring MVC, només resta crear el controlador que manegarà les peticions.

El *Controlador* és el responsable de processar les peticions, construir el model apropiat i passar-lo com a resposta a la vista que calgui.

Creeu el paquet *cat.xtec.ioc* i a dins la classe Java *HolaController* amb el contingut que es mostra a continuació.

```

1 package cat.xtec.ioc;
2
3 import java.io.IOException;
4 import javax.servlet.ServletException;
5 import javax.servlet.http.HttpServletRequest;
6 import javax.servlet.http.HttpServletResponse;
7 import org.springframework.stereotype.Controller;
8 import org.springframework.ui.ModelMap;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RequestMethod;
11 import org.springframework.web.servlet.ModelAndView;
12
13 @Controller
14 @RequestMapping("/")
15 public class HolaController {
16
17     @RequestMapping(method = RequestMethod.GET)
18     public ModelAndView handleRequest(HttpServletRequest request,
19                                         HttpServletResponse response)
20             throws ServletException, IOException {
21         ModelAndView modelview = new ModelAndView("resposta");
22         modelview.getModelMap().addAttribute("salutacio", "Hola a tothom");
23         return modelview;
24     }
25
26     @RequestMapping(value = "/bondia", method = RequestMethod.GET)
27     public ModelAndView handleRequestDia(HttpServletRequest request,
28                                         HttpServletResponse response)
29             throws ServletException, IOException {
30         ModelAndView modelview = new ModelAndView("resposta");
31         modelview.getModelMap().addAttribute("salutacio", "Bon dia a tothom");
32         return modelview;
33     }
34
35     @RequestMapping(value = "/bonanit", method = RequestMethod.GET)
36     public ModelAndView handleRequestNit(HttpServletRequest request,
37                                         HttpServletResponse response)
38             throws ServletException, IOException {
39         ModelAndView modelview = new ModelAndView("resposta");
40         modelview.getModelMap().addAttribute("salutacio", "Bona nit a tothom");
41     }
42 }
```

L'anotació `@Controller` és molt bàsica, però suficient en aquesta implementació.

L'anotació `@RequestMapping` serveix per determinar l'origen de la petició i així poder decidir quin mètode manegarà el retorn. Com podeu veure, es pot fer a nivell de la classe o a nivell de mètode. Els paràmetres d'aquesta anotació els podeu consultar a la documentació de referència de Spring; en aquest cas hem fet servir el tipus de petició i l'URL.

Cada mètode encarregat (*handler*) de construir el retorn crea un nou objecte *ModelAndView* amb el nom de la vista que ha de retornar. Aquest

nom, en combinació amb la configuració feta a DispatcherServlet-servlet.xml (InternalResourceViewResolver), serveix perquè Spring MVC determini exactament la vista (WEB-INF/views/resposta.jsp).

Als mètodes esmentats, abans de retornar la vista, s'afegeix un atribut al model (*ModelMap*) que en el nostre exemple serveix per provar diferents orígens a les peticions i veure diferents respostes.

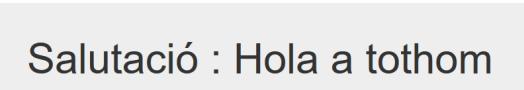
Finalment, creeu la carpeta *views* dins de *WEB-INF*, i a dins el fitxer *resposta.jsp* amb el contingut que es mostra a continuació.

```

1 <%@ page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3 <html lang="ca">
4   <head>
5     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/
7       bootstrap.min.css">
8     <title>Salutació</title>
9   </head>
10  <body>
11    <section>
12      <div class="jumbotron">
13        <div class="container">
14          <h1>Salutació : ${salutacio} </h1>
15        </div>
16      </div>
17    </section>
18  </body>
</html>
```

En aquest cas, quan es mostri la vista al navegador, el valor de la variable *salutacio* canviàrà en funció de l'URL de petició. Executeu l'aplicació sense més, i us ha de donar la salutació estàndard (vegeu la figura 1.13).

FIGURA 1.13. Sortida de l'aplicació “Hola, Món” amb Spring MVC



Salutació : Hola a tothom

Si al navegador canvieu l'URL afegint-hi */bondia* o */bonanit*, les salutacions canviaran. Proveu-ho.

1.3.5 Conclusions del nostre ‘Hola, Món’ Spring MVC

L'aplicació simple Hola, Món està creada a partir d'una petició des d'un URL al navegador mostra diferents missatges. L'estructura està feta amb Maven WebApp, però després hi ha les dependències necessàries per a J2EE i Spring MVC al fitxer pom.xml.

Una aplicació Spring MVC té una arquitectura de classes i un procés determinat. Tota petició és rebuda per un Front Controller (Dispatcher Servlet) confi-

gurat a web.xml. Aquest la dirigeix cap al controlador que realment la manegarà (*handler*), però en realitat és el controlador (*HolaController*, en el nostre exemple) qui mitjançant les anotacions agafarà les peticions, construirà un *ModelAndView* i el retornarà al Front Controller per tal que aquest el retorne al client (el navegador).

Per tal que tot funcioni correctament hem configurat la resolució dinàmica de les anotacions i la resolució de la vista efectiva mitjançant el fitxer DispatcherServlet-servlet.xml.

1.4 Estoc de medicaments, benvinguda

Desenvoluparem la pàgina d'inici (benvinguda) d'una aplicació per gestionar **estocs de medicaments** que anomenarem “**stmedioc**”. Crearem el projecte amb l'arquitectura d'una aplicació web basada en Spring MVC i amb la configuració basada en XML.

Configuració dels 'beans'

La configuració dels *beans* pot ser definida via fitxers de configuració XML, com applicationContext.xml i també via classes de configuració Java (*JavaConfig*).

1.4.1 Creació i configuració inicial del projecte Estoc de medicaments

És convenient fer servir el projecte anterior per consultar, però és millor anar constraint el vostre propi projecte amb els indicacions que es van donant. No obstant això, si voleu executar el projecte sencer segurament us obliga a fer *clean and build* abans.

Creeu un nou projecte *Maven/Project from Archetype* amb l'arquetipus maven-archetype-webapp i anomeneu-lo “stmedioc101”.

Des de la pestanya *Files* de NetBeans o des del mateix explorador de fitxers del sistema operatiu, creeu la carpeta *Java* dins de la carpeta *src/main*. És important fer aquest pas, perquè veureu Source Packages a la pestanya de *Projects*.

Canvieu el fitxer de configuració web.xml.

```

1 <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
4                           http://java.sun.com/xml/ns/javaee/web-app_3_0.
5                           xsd">
6     <display-name>Estoc de Medicaments</display-name>
7   </web-app>
```

A la secció *Dependències* de pom.xml heu d'afegir les següents dependències:

```

1 <dependency>
2   <groupId>org.springframework</groupId>
3   <artifactId>spring-webmvc</artifactId>
4   <version>4.0.3.RELEASE</version>
```

```

5      </dependency>
6      <dependency>
7          <groupId>javax.servlet</groupId>
8          <artifactId>jstl</artifactId>
9          <version>1.2</version>
10     </dependency>
11     <dependency>
12         <groupId>javax.servlet</groupId>
13         <artifactId>javax.servlet-api</artifactId>
14         <version>3.1.0</version>
15     </dependency>
16     <dependency>
17         <groupId>javax</groupId>
18         <artifactId>javaee-web-api</artifactId>
19         <version>7.0</version>
20     </dependency>

```

Afegiu la propietat que defineix la utilització de la codificació UTF-8 en el mateix fitxer pom.xml.

```

1 <properties>
2     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3 </properties>

```

Executeu l'aplicació seleccionant el servidor **Glassfish** i fent que recordi aquesta decisió permanentment.

1.4.2 Pàgina de benvinguda

Creareu una pàgina de benvinguda per a la vostra aplicació consistent a mostrar un bàner però amb el procés de Spring MVC.

Per fer això haureu de crear la pàgina jsp, configurar el Dispatcher Servlet i crear el controlador per manegar la petició de benvinguda.

Creeu el fitxer DispatcherServlet-servlet.xml en una nova carpeta de nom *spring* dins de WEB-INF. El contingut és el que es mostra a continuació.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7                           http://www.springframework.org/schema/beans/spring-
8                           beans.xsd
9                           http://www.springframework.org/schema/context
10                          http://www.springframework.org/schema/context/spring-
11                          context-4.0.xsd
12                          http://www.springframework.org/schema/mvc
13                          http://www.springframework.org/schema/mvc/spring-mvc-
14                          4.0.xsd">
15
16     <mvc:annotation-driven />
17     <context:component-scan base-package="cat.xtec.ioc.controller" />
18
19     <bean class="org.springframework.web.servlet.view.
20           InternalResourceViewResolver">
21         <property name="prefix" value="/WEB-INF/views/" />
22     </bean>

```

```

18      <property name="suffix" value=".jsp" />
19  </bean>
20 </beans>
```

En aquesta configuració es podrien tenir més paquets en *base-package* simplement separant-los amb comes.

Definim el **Dispatcher Servlet** (Front Controller) afegint el següent contingut al fitxer de configuració web.xml:

```

1 <servlet>
2   <servlet-name>DispatcherServlet</servlet-name>
3   <servlet-class>org.springframework.web.servlet.DispatcherServlet</
4     servlet-class>
5   <init-param>
6     <param-name>contextConfigLocation</param-name>
7     <param-value>/WEB-INF/spring/DispatcherServlet-servlet.xml</param-
8       value>
9   </init-param>
10  <load-on-startup>1</load-on-startup>
11  </servlet>
12  <servlet-mapping>
13    <servlet-name>DispatcherServlet</servlet-name>
14    <url-pattern>/</url-pattern>
15  </servlet-mapping>
```

Pel que fa al controlador que gestionarà les peticions, creeu el paquet **cat.xtec.ioc.controller** i a dins la classe Java HomeController amb el contingut que es mostra a continuació:

```

1 package cat.xtec.ioc.controller;
2
3 import java.io.IOException;
4 import javax.servlet.ServletException;
5 import javax.servlet.http.HttpServletRequest;
6 import javax.servlet.http.HttpServletResponse;
7 import org.springframework.stereotype.Controller;
8 import org.springframework.web.bind.annotation.RequestMapping;
9 import org.springframework.web.bind.annotation.RequestMethod;
10 import org.springframework.web.servlet.ModelAndView;
11
12 @Controller
13 public class HomeController {
14
15     @RequestMapping(value = "/", method = RequestMethod.GET)
16     public ModelAndView handleRequest(HttpServletRequest request,
17         HttpServletResponse response)
18         throws ServletException, IOException {
19         ModelAndView modelview = new ModelAndView("welcome");
20         modelview.getModelMap().addAttribute("benvinguda", "Benvingut Estoc de
21             Medicaments!");
22         modelview.getModelMap().addAttribute("tagline", "Una aplicació de l'
23             Institut Obert de Catalunya");
24         return modelview;
25     }
26 }
```

Finalment, creeu la carpeta *views* dins de *WEB-INF* i a dins el fitxer welcome.jsp amb el contingut que es mostra a continuació.

```

1 <%@ page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
```

```
3 <html lang="ca">
4   <head>
5     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     <link rel="stylesheet" href="/netdna.bootstrapcdn.com/bootstrap/3.0.0/css/
7       bootstrap.min.css">
8     <title>Welcome</title>
9   </head>
10  <body>
11    <section>
12      <div class="jumbotron">
13        <div class="container">
14          <h1> ${benvinguda} </h1>
15          <p> ${tagline} </p>
16        </div>
17      </div>
18    </section>
19  </body>
</html>
```

En executar l'aplicació us ha d'aparèixer el bàner que es mostra en la figura 1.14.

FIGURA 1.14. Sortida de l'aplicació. “Estoc de medicaments”, benvinguda

Benvingut Estoc de Medicaments!

Una aplicació de l'Institut Obert de Catalunya

Ara ja teniu el projecte amb els elements i la configuració bàsica per continuar desenvolupant el vostre estoc de medicaments.

1.5 Què s'ha après?

En aquest apartat hem fet servir **Maven** com a eina que dóna estructura als projectes i permet baixar les llibreries que cal fer servir des de repositoris centrals a partir de la configuració de **dependències**. El gran avantatge de Maven és fer independent el nostre desenvolupament de l'IDE que es fa servir.

Hem introduït el procés **petició-resposta** de Spring MVC, així com a l'arquitectura que en dóna suport. Heu creat dues aplicacions Spring MVC amb la mateixa estructura, és a dir, una part de vistes .jsp, la configuració del Dispatcher Servlet (o Front Controller) i la classe que fa de controlador per manegar les peticions i que retorna l'objecte ModelAndView.

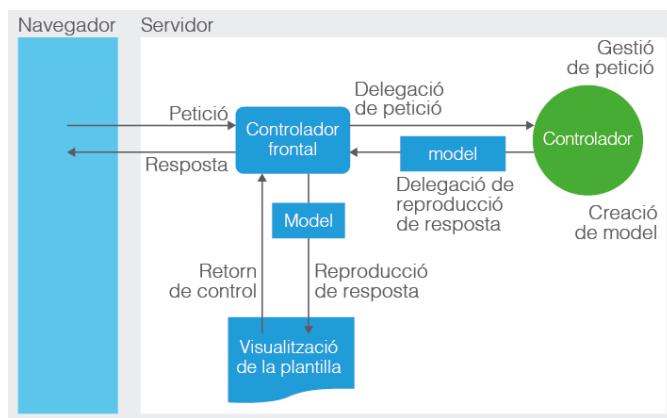
Les aplicacions fets fins ara en aquesta unitat només mostren un text variable o no, encara que amb l'arquitectura de Spring MVC. Hem d'avanscar una mica més i donar més funcionalitat a “Estoc de medicaments”.

En els apartats següents veurem com afegir aquesta funcionalitat de manera ordenada, creant els paquets i les classes amb els patrons que suggereix Spring MVC.

2. Spring MVC, aplicació web

L'arquitectura i el procés d'Spring MVC respecte a una petició des d'un navegador és la que es mostra a la figura 2.1.

FIGURA 2.1. Arquitectura i procés d'una aplicació Spring MVC



La petició feta des d'un navegador client és recollida pel Front Controller (Dispatcher Servlet) i la passa al controlador adient. Aquest construeix el **model** i el retorna (amb l'estat que correspongui) al Dispatcher Servlet que retornarà la resposta, determinant la vista a retornar a partir del View Resolver assignat i amb els valors del model.

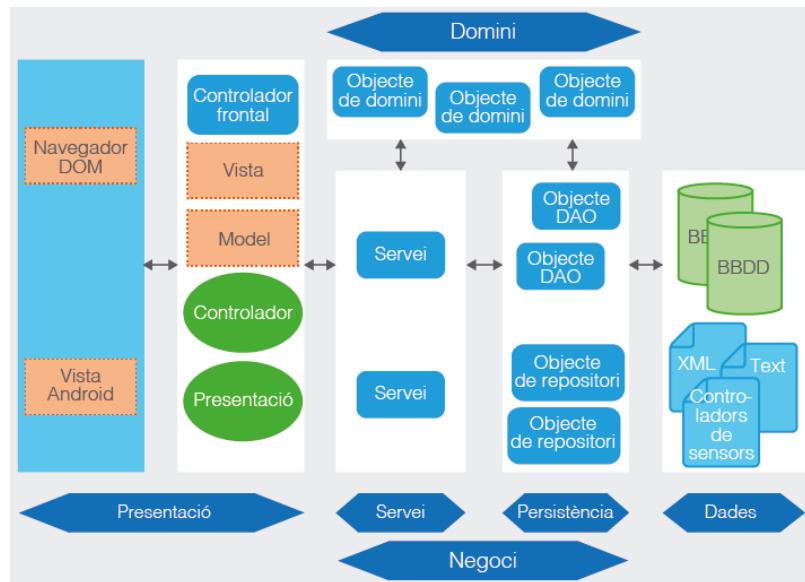
Veurem com el controlador construeix el model, és a dir, la part de negoci de la vostra aplicació web (*enterprise-level*).

Amb Spring MVC podeu fer servir una bona pràctica per a aquest tipus de desenvolupament, consistent a estructurar el codi en capes (*layers*) i donant reusabilitat i baix acoblament a la vostra aplicació.

Les capes que es recomanen són quatre:

- presentació (*Presentation*)
- domini (*Domain*)
- serveis (*Services*)
- persistència (*Persist*)

El diagrama de la figura 2.2 mostra aquestes capes i les seves interaccions.

FIGURA 2.2. Capes i interaccions d'una aplicació Spring MVC

Els objectes vistos fins ara, com Dispatcher Servlet, controladors, View Resolvers i d'altres conformen la capa de **presentació**.

La capa de **persistència** és la que conté els objectes que interaccionen amb les dades; per exemple, obtenint un conjunt de registres d'una base de dades a partir d'una consulta SQL.

Un controlador podria demanar directament les dades a la capa de persistència, però a l'arquitectura del diagrama es proposa la creació de la capa de **servei** per tal de poder aplicar les regles de negocis amb i/o abans d'obtenir les dades. Per exemple, l'aplicació d'una restricció no implementada a la base de dades, com no deixar a un client fer una comanda si el seu deute és superior a cert import.

En tot cas, les dades que s'obtenen o s'estan construint es mapegen sobre els objectes de la capa de **domini**; per exemple, objectes de classes entitat que poden representar una taula d'una base de dades.

Continuareu amb el vostre projecte d'Estoc de medicaments (“stmedioc”) desenvolupant la resta de capes i descrivint els conceptes i la metodologia implicades en cadascuna d'elles.

2.1 Estoc de medicaments, capa de domini

La capa **domini** d'una aplicació web consisteix en la implementació de les classes d'un o més models de domini.

El **model de domini** és la representació, per exemple amb un diagrama UML, de les classes corresponents a les dades del problema a resoldre de la lògica de negocis.

En el cas de l'aplicació d'Estoc de medicaments, dissenyada amb propòsits pedagògics, només hi ha una entitat Medicament i, per tant, la capa de domini només tindrà una classe, que anomenareu Medicament (vegeu la figura 2.3).

FIGURA 2.3. Classe Medicament

```
Medicament

- medicamentId: String
- name: String
- price: double
- description: String
- producer: String
- category: String
- stockQuantity: long
- stockInOrder: long
- active: boolean
+ Medicament()
+ Medicament (medicamentId: String, name: String,
  price: double)
+ toString(): String
```

Un entitat Medicament podria contenir més propietats i mètodes, però de moment només fareu servir aquest a la vostra aplicació. El nom de les propietats és suficient per relacionar-les amb els conceptes que representen. No obstant això, cal fer algunes apreciacions:

- *medicamentID* és el codi de medicament.
- *producer* és el proveïdor.
- *stockQuantity* són les unitats emmagatzemades i *stockInOrder* són les unitats que estan demanades al proveïdor però encara no heu introduït al magatzem.
- *active* és per indicar si un medicament es fa servir (valor *true*) o s'ha donat de baixa (valor *false*).

Els objectes creats amb classes de domini s'acostumen a dir **objectes de domini** (*Domain Object*).

2.1.1 Creació del domini

Com que el nostre domini està format únicament per la classe Medicament, al projecte “stmedioc” heu de crear un paquet `ioc.xtec.cat.domain` i la classe Medicament amb el contingut que es mostra a continuació:

```
1 package cat.xtec.ioc.domain;
2
3 public class Medicament {
4
5     private String medicamentId;
6     private String name;
```

El codi del projecte “stmedioc” en l'estat de capa de domini es pot descarregar des de l'enllaç que trobareu als annexos de la unitat. Però per seguir el desenvolupament de la capa de domini és millor partir del que ja heu fet servir per a Estoc de medicaments. Benvinguda i copiar-lo amb el nom de “stmedioc201”.

```

7     private double price;
8     private String description;
9     private String producer;
10    private String category;
11    private long stockQuantity;
12    private long stockInOrder;
13    private boolean active;
14
15    public Medicament() {
16        super();
17    }
18
19    public Medicament(String medicamentId, String name, double price) {
20        this.medicamentId = medicamentId;
21        this.name = name;
22        this.price = price;
23    }
24
25    //Heu d'afegir els getters i setters de totes les propietats
26
27    @Override
28    public boolean equals(Object obj) {
29        if (this == obj) {
30            return true;
31        }
32        if (obj == null) {
33            return false;
34        }
35        if (getClass() != obj.getClass()) {
36            return false;
37        }
38        Medicament other = (Medicament) obj;
39        if (medicamentId == null) {
40            if (other.medicamentId != null) {
41                return false;
42            }
43        } else if (!medicamentId.equals(other.medicamentId)) {
44            return false;
45        }
46        return true;
47    }
48
49    @Override
50    public int hashCode() {
51        final int prime = 31;
52        int result = 1;
53        result = prime * result
54            + ((medicamentId == null) ? 0 : medicamentId.hashCode());
55        return result;
56    }
57
58    @Override
59    public String toString() {
60        return "Medicament [codi=" + medicamentId + ", nom=" + name + "]";
61    }
62 }
```

2.1.2 Fent servir el domini

Penseu ara en la necessitat de mostrar per al navegador un medicament mitjançant una petició GET des d'un URL.

Davant d'una petició, el Dispatcher Servlet delegarà a un controlador la petició i aquest s'encarregarà de construir el model.

A Spring MVC, amb l'estructura de capes que esteu desenvolupant, implementareu un nou controlador `MedicamentController` que agafi aquesta petició i construeixi el model a partir del domini (de la classe `Medicament`).

Heu de crear la classe `MedicamentController` dins del paquet `cat.xtec.ioc.controller` amb el contingut que es mostra a continuació.

```

1 package cat.xtec.ioc.controller;
2
3 import cat.xtec.ioc.domain.Medicament;
4 import java.io.IOException;
5 import javax.servlet.ServletException;
6 import javax.servlet.http.HttpServletRequest;
7 import javax.servlet.http.HttpServletResponse;
8 import org.springframework.stereotype.Controller;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RequestMethod;
11 import org.springframework.web.servlet.ModelAndView;
12
13 @Controller
14 public class MedicamentController {
15
16     @RequestMapping(value = "/medicaments", method = RequestMethod.GET)
17     public ModelAndView handleRequest(HttpServletRequest request,
18         HttpServletResponse response)
19             throws ServletException, IOException {
20         ModelAndView modelview = new ModelAndView("medicaments");
21         Medicament ibuprofe = new Medicament("M010", "Ibuprofè", 2);
22         ibuprofe.setDescription("Ibuprofè de 600mg");
23         ibuprofe.setCategory("Anti-inflamatori");
24         ibuprofe.setProducer("Cinfa");
25         ibuprofe.setStockQuantity(214);
26         modelview.getModelMap().addAttribute("medicament", ibuprofe);
27         return modelview;
28     }
29 }
```

Cal destacar que al controlador `MedicamentController` esteu donant valors a l'entitat, i en el model de capes això serà responsabilitat d'altres. En aquest cas, doneu els valors en el controlador perquè encara no heu desenvolupat les altres capes, però després canviareu aquesta classe per fer-ho on toca.

Fixeu-vos que fins ara els atributs que heu afegit al model eren simples, com *string*. En el cas de `MedicamentController` esteu creant un atribut que és un objecte: un *domain object* `Medicament`.

`MedicamentController` retorna un `ModelAndView` amb nom *medicaments*. Segons la configuració que havíeu fet per al View Resolver a `DispatcherServlet-servlet.xml`, Spring cercrà la vista `WEB-INF/views/medicaments.jsp` per mostrar-la i afegirà l'atribut `medicament` (objecte `Medicament`) a la resposta.

Heu de crear aquesta vista a la carpeta esmentada amb el contingut que es mostra a continuació.

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3 <html lang="ca">
4     <head>
5         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6         <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/
7             css/bootstrap.min.css">
8         <title>Medicaments</title>
```

```

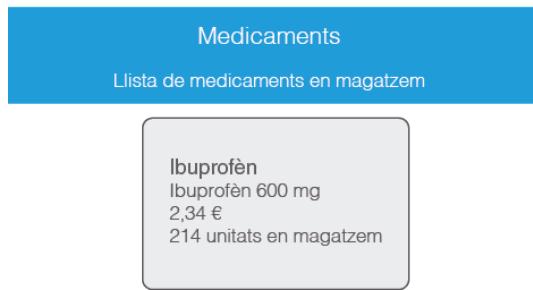
8   </head>
9   <body>
10  <section>
11    <div class="jumbotron">
12      <div class="container">
13        <h1>Medicaments</h1>
14        <p>Llista de medicaments en magatzem</p>
15      </div>
16    </div>
17  </section>
18  <section class="container">
19    <div class="row">
20      <div class="col-sm-6 col-md-3" style="padding-bottom: 15px">
21        <div class="thumbnail">
22          <div class="caption">
23            <h3>${medicament.name}</h3>
24            <p>${medicament.description}</p>
25            <p>${medicament.price} €</p>
26            <p>Hi ha ${medicament.stockQuantity} unitats en
27              magatzem</p>
28          </div>
29        </div>
30      </div>
31    </div>
32  </body>
33 </html>

```

Fixeu-vos que esteu servint les expressions tipus EL (*Expression Language*) \${atribut.propietat} i que com a atribut havíeu passat un objecte Medicament, i per tant podeu usar les seves propietats. Tingueu en compte que, per exemple, amb \${medicament.name} s'executarà el mètode medicament.getName(), i si no l'heu implementat (*getters and setters*) llavors el resultat serà *null*.

Si executeu l'aplicació i afegiu a l'URL del navegador /medicaments podreu veure el que es mostra en la figura 2.4.

FIGURA 2.4. Sortida de l'aplicació Estoc de medicaments, domini



Partíeu de la capa de presentació feta, però anireu afegint controladors i vistes a mesura que les necessiteu.

La capa de **domini** es correspon pràcticament amb les classes Entitat del problema a resoldre, i en el cas de l'aplicació Estoc de medicaments és la classe Medicament. Però heu fet trampes posant valors en el controlador MedicamentController amb la finalitat de provar. I això s'ha de resoldre continuant amb el desenvolupament de les següents capes.

2.2 Estoc de medicaments, capa de persistència

La capa de **persistència** és el conjunt d'objectes que interaccionen amb les dades. Les dades poden estar emmagatzemades en bases de dades i en altres tipus de formats, com XML, JSON, imatges, vídeo, etc.

La capa de persistència conté objectes **repositori** (*Repository Objects*) que permeten mapejar les dades de la font de dades (base de dades o no) amb els objectes de domini (*Domain Object*). Són, en realitat, els responsables de les operacions *CRUD* (*Create, Read, Update i Delete*).

En operacions d'obtenció de dades, els objectes repositori consulten la font de dades, per exemple via SQL, i el resultat és mapejat sobre objectes de domini.

En operacions d'actualització de les dades, els objectes repositori parteixen de l'estat dels objectes de domini i amb els seus valors actualitzen la font de dades.

Per indicar a Spring que una classe és un repositori es fa servir l'anotació `@Repository` (`org.springframework.stereotype.Repository`). Aquesta anotació també permet que les excepcions SQL es converteixin en `DataAccessExceptions` de Spring.

Amb aquests conceptes descrits ja podeu crear la capa de persistència de la vostra aplicació Estoc de medicaments.

Al desenvolupament de la capa de domini heu mostrat un únic medicament que, a més, temporalment, es crea directament al controlador. Mostrareu una llista de medicaments i a més traureu la creació del medicament del controlador.

La manera d'accendir a les dades està fora de l'abast d'aquesta unitat, i per això fareu servir objectes en memòria com si fossin la base de dades. Només haureu de canviar aquests objectes per d'altres que realment es connectin i dialoguin amb la base de dades si voleu dotar de veritable persistència l'Estoc de medicaments.

Creareu una interfície `MedicamentRepository` i la implementareu amb la classe `InMemoryMedicamentRepository` amb els mètodes per mostrar la llista de medicaments, només un mètode en aquest cas. Per altra banda, fareu que el controlador dialogui directament amb aquesta capa de persistència, encara que ho canviareu més endavant per tal que dialogui amb la capa de servei.

Feu servir una interfície, perquè això us permet que el controlador declari i cridi la interfície, i d'aquesta manera podeu canviar la implementació sense canviar res més.

2.2.1 Creació de la persistència

Afegiu el paquet *cat.ioc.xtec.domain.repository* al nostre projecte “stmedioc”, i en aquest paquet una nova interfície MedicamentRepository amb el contingut següent:

```

1 package cat.xtec.ioc.domain.repository;
2
3 import cat.xtec.ioc.domain.Medicament;
4 import java.util.List;
5
6 public interface MedicamentRepository {
7     List <Medicament> getAllMedicaments();
8 }
```

El codi del projecte “stmedioc” en l'estat de capa de persistència es pot descarregar des de l'enllaç que trobareu als annexos de la unitat.

Però per seguir el desenvolupament de la capa de persistència és millor partir del que ja heu fet servir per a Estoc de medicaments, capa de domini, i copiar-lo amb el nom de “stmedioc202”.

Afegiu el paquet *cat.ioc.xtec.domain.repository.impl* al nostre projecte “stmedioc”. També us interessa que Spring cerqui automàticament les classes d'aquest paquet per poder injectar els objectes quan es trobi una anotació d'aquest tipus. Per això, canvieu la propietat *context:component-scan* de DispatcherServlet-servlet.xml:

```

1 <context:component-scan base-package="cat.xtec.ioc.controller cat.xtec.ioc.
domain.repository" />
```

En el paquet *cat.ioc.xtec.domain.repository.impl*, la classe *InMemoryMedicamentRepository* implementa *MedicamentRepository* amb el següent contingut:

```

1 package cat.xtec.ioc.domain.repository.impl;
2
3 import cat.xtec.ioc.domain.Medicament;
4 import cat.xtec.ioc.domain.repository.MedicamentRepository;
5 import java.util.ArrayList;
6 import java.util.List;
7 import org.springframework.stereotype.Repository;
8
9 @Repository
10 public class InMemoryMedicamentRepository implements MedicamentRepository {
11
12     private List<Medicament> listOfMedicaments = new ArrayList<Medicament>();
13
14     public InMemoryMedicamentRepository() {
15         Medicament ibuprofe = new Medicament("M010", "Ibuprofè", 2);
16             ibuprofe.setDescription("Ibuprofè de 600mg");
17             ibuprofe.setCategory("Anti-inflamatori");
18             ibuprofe.setProducer("Cinfa");
19             ibuprofe.setStockQuantity(214);
20
21         Medicament paracetamol = new Medicament("M020", "Paracetamol", 2.6);
22             paracetamol.setDescription("Paracetamol 1g");
23             paracetamol.setCategory("Afgèsic");
24             paracetamol.setProducer("Ferrer");
25             paracetamol.setStockQuantity(56);
26
27         Medicament acacetilsalicilico = new Medicament("M030", "Ac Acetil Salicí
28             lico", 2.6);
29             acacetilsalicilico.setDescription("Ac Acetil Salicílico");
30             acacetilsalicilico.setCategory("Afgèsic");
31             acacetilsalicilico.setProducer("Bayer");
32             acacetilsalicilico.setStockQuantity(15);
```

```

32     listOfMedicaments.add(ibuprofe);
33     listOfMedicaments.add(paracetamol);
34     listOfMedicaments.add(acacetilsalicilico);
35 }
36
37 public List<Medicament> getAllMedicaments() {
38     return listOfMedicaments;
39 }
40 }
41 }
```

Aquesta classe repository hauria de connectar amb la font de dades, però com que és fora de l'abast d'aquesta unitat feu que les dades resideixin en memòria creant-les en el constructor. Per això, `getAllMedicaments` retorna la llista de medicaments propietat de la classe, però en realitat hauria d'anar a buscar-la a la font de dades.

Canvieu el controlador `MedicamentController` per obtenir la llista de medicaments des de la capa de persistència. El contingut de `MedicamentController` sense els imports és com es mostra a continuació:

```

1 @Controller
2 public class MedicamentController {
3
4     @Autowired
5     private MedicamentRepository medicamentRepository;
6
7     @RequestMapping(value = "/medicaments", method = RequestMethod.GET)
8     public ModelAndView handleRequest(HttpServletRequest request,
9             HttpServletResponse response)
10            throws ServletException, IOException {
11         ModelAndView modelview = new ModelAndView("medicaments");
12         modelview.getModelMap().addAttribute("medicaments",
13             medicamentRepository.getAllMedicaments());
14         return modelview;
15     }
16 }
```

Haureu d'afegir-hi els imports:

```

1 import org.springframework.beans.factory.annotation.Autowired;
2 import cat.xtec.ioc.domain.repository.MedicamentRepository;
```

Hi podeu suprimir l'import:

```

1 import cat.xtec.ioc.domain.Medicament;
```

S'ha creat la propietat `medicamentRepository` que, mitjançant l'anotació `@Autowired`, serà inicialitzada per Spring amb l'objecte de tipus `MedicamentRespository` del Web Application Context.

S'ha canviat la creació d'un únic medicament en la mateixa classe controlador per a la crida a la llista de tots els medicaments que us proporciona la persistència. Noteu que s'ha canviat l'atribut `medicament` per `medicaments`.

Fixeu-vos com no es crida directament la classe `InMemoryMedicamentRepository`, ja que es declara i es crida la interfície.

Això us permetrà canvis en la font de dades i en la manera de dialogar amb ella sense tocar el controlador.

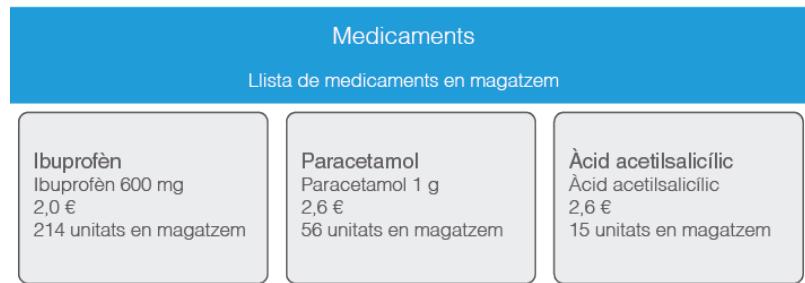
Només us queda mostrar la llista dels medicaments que hi ha a la vostra font de dades (creats en memòria). Canvieu la vista medicaments.jsp: heu d'afegir al principi de tot la declaració del *taglib* i heu de canviar tot el contingut de la *div* amb *classe='row'* pel codi que es mostra a continuació.

```

1 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2 ...
3     <c:forEach items="${medicaments}" var="medicament">
4         <div class="col-sm-6 col-md-3" style="padding-bottom: 15px
5             ">
6             <div class="thumbnail">
7                 <div class="caption">
8                     <h3>${medicament.name}</h3>
9                     <p>${medicament.description}</p>
10                    <p>${medicament.price}</p>
11                    <p>Hi ha ${medicament.stockQuantity} unitats en
12                      magatzem</p>
13                </div>
14            </div>
15        </c:forEach>
```

Executeu l'aplicació i afegiu */medicaments* a l'URL. Heu d'obtenir un resultat similar al de la figura 2.5.

FIGURA 2.5. Sortida de l'aplicació Estoc de medicaments, persistència



L'URL */medicaments* ha fet que el Dispatcher Servlet passi la petició a MedicamentController. En primer lloc, demana la injecció d'un objecte MedicamentRepository (anotació `@Autowired`) i Spring retorna un objecte de la implementació InMemoryMedicamentRepository.

Recordeu que feu servir una classe que construeix valors ficticis en memòria perquè l'accés a dades no és l'objectiu d'aquesta unitat, però podríeu fer una classe que retornarà els valors des d'una base de dades, per exemple MedicamentDAO, que implementaria MedicamentRepository i no hauríeu de canviar MedicamentController. Això és l'acoblament feble entre capes.

En segon lloc, es construeix el ModelAndView a partir del repositori, demanant tots els medicaments al repositori i passant-los a la vista Medicaments sobre l'atribut del mateix nom.

Finalment, es mostra la vista que, amb etiquetes de JSTL, recorrerà els objectes de l'atribut `medicaments` i els mostrerà.

2.3 Estoc de medicaments, capa de servei

El projecte Estoc de medicaments conté la capa de presentació, amb les vistes i els controladors; la capa de domini, amb la representació d'un medicament, i la capa de persistència, que dialoga amb les dades mitjançant objectes `repository`.

En el projecte actual, els controladors criden directament, via injecció, els objectes `repository`. No obstant això, si recupereu el diagrama d'arquitectura i procés d'una aplicació Spring MVC, els controladors només dialoguen amb la capa de servei que us falta desenvolupar.

Els objectes `repository` fan operacions *CRUD* simples, i els resultats els poden emmagatzemar en memòria en forma d'objectes de domini. Heu vist que obtenguïu una llista de medicaments perquè està guardada en una *List* d'objectes `Medicament` (objectes de domini) al repositori.

Però on posareu operacions que siguin més complexes que abastin més d'una operació *CRUD*, més d'una entitat, o simplement que afegeixin restriccions? És a dir, on posareu les regles de negoci?

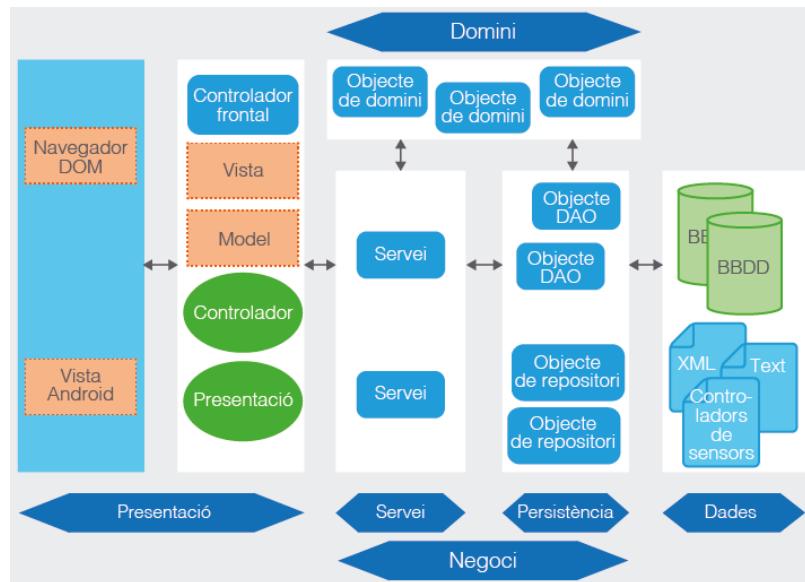
Per exemple, penseu en un moviment d'estoc del nostre medicament que signifiqui baixar 10 unitats d'ibuprofèn. En una primera aproximació podríeu dir que això consisteix a crear un mètode al repositori que faci un *update* (com és a memòria, seria actualitzar l'element adient a `List<Medicament> listOfMedicaments` de `InMemoryMedicamentRepository`).

Però hauríeu de fer més operacions. Hauríeu de veure que el medicament que es vol actualitzar existeix, que té suficient estoc, i després fer efectiva l'actualització. No ho fareu per ara, però a tot això hauríeu d'afegir-li la comprovació de si l'usuari té autorització per fer aquesta operació.

Per contenir aquestes operacions complexes que abasten més d'una operació simple, on podeu fer servir un o més objectes de domini, fareu servir la capa de **servei**.

Com podeu veure en apartats d'accés a dades, les transaccions es defineixen a la **capa de servei**.

En el projecte **Estoc de medicaments** que esteu desenvolupant creareu un servei que faci els moviments d'estoc sobre els medicaments, és a dir, que simplement incrementi o redueixi les unitats en estoc. Llavors, la crida de la capa de presentació es farà com preveu l'arquitectura i el procés d'una aplicació web amb Spring MVC, és a dir, com es mostra en la figura 2.6.

FIGURA 2.6. Capes i interaccions a Spring MVC

Amb aquest objectiu, creareu un controlador específic que atengui les peticions de moviments d'estoc que cridi el vostre servei i que retorni el resultat. El servei cridrà el repositori i aquest farà l'actualització. Com és a memòria, farà l'actualització sobre la *List* de medicaments.

Però al repositori només hi ha un mètode que retorna la llista de tots els medicaments. Llavors heu de modificar el repositori per adaptar-lo a les operacions que havíeu enumerat:

- El medicament que es vol actualitzar existeix.
- El medicament té suficient estoc (només si és un moviment de decrement).
- Fer efectiva l'actualització.

Heu d'afegir al `repository` els mètodes adients per resoldre les operacions.

El codi del projecte “stmedioc” en l'estat de capa de servei es pot descarregar del següent .

Però per seguir el desenvolupament de la capa de servei és millor partir del que ja heu fet servir per a Estoc de medicaments, capa de persistència, i copiar-lo amb el nom de “stmedioc203” del següent .

2.3.1 Adaptació del repositori

Heu d'accendir a un medicament concret a partir del seu codi (`medicamentId`), i això us permetrà comprovar que existeix i veure si es pot fer el moviment d'estoc (ha de tenir prou unitats si voleu treure estoc).

Modifiqueu la interfície MedicamentRepository afegint-hi el mètode Medicament getMedicamentById(String medicamentId);.

Implementeu el mètode a InMemoryMedicamentRepository tal com es mostra a continuació:

```

1 public Medicament getMedicamentById(String medicamentId) {
2     Medicament medicamentById = null;
3     for (Medicament medicament : listOfMedicaments) {
4         if (medicament != null && medicament.getMedicamentId() != null
5             && medicament.getMedicamentId().equals(medicamentId)) {
6             medicamentById = medicament;
7             break;
8         }
9     }
10    if (medicamentById == null) {
11        throw new IllegalArgumentException(
12            "No s'han trobat medicaments amb el codi: " + medicamentId)
13            ;
14    }
15    return medicamentById;
}

```

El mètode getMedicamentById cerca el medicament que coincideix en *id* amb el codi que es passa pel paràmetre. Fixeu-vos que si el troba retorna l'objecte de domini Medicament, i en un altre cas, llança una excepció.

2.3.2 Creació del servei

Heu de crear el servei que faci totes les operacions per aconseguir l'actualització de l'estoc comprovant les regles de negoci: existeix el medicament i no deixeu l'estoc en negatiu.

Com en altres casos fareu servir una interfície, ja que us permet l'acoblament més feble entre les capes.

Creeu un nou paquet cat.xtec.ioc.service i en ell la interfície MovimentStockService amb el codi que es mostra a continuació:

```

1 package cat.xtec.ioc.service;
2
3 public interface MovimentStockService {
4     void processMovimentStock(String medicamentId, long quantity, int signe);
5 }

```

Creeu un nou paquet cat.xtec.ioc.service.impl i també la implementació MovimentStockServiceImpl amb el codi que es mostra a continuació:

```

1 package cat.xtec.ioc.service.impl;
2
3 import cat.xtec.ioc.domain.Medicament;
4 import cat.xtec.ioc.domain.repository.MedicamentRepository;
5 import cat.xtec.ioc.service.MovimentStockService;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.stereotype.Service;
8

```

```

9  @Service
10 public class MovimentStockServiceImpl implements MovimentStockService {
11
12     @Autowired
13     private MedicamentRepository medicamentRepository;
14
15     public void processMovimentStock(String medicamentId, long quantity, int
16         signe) {
17         Medicament medicamentById = medicamentRepository.getMedicamentById(
18             medicamentId);
19         long signedQuantity = quantity * signe;
20         if ((medicamentById.getStockQuantity() + signedQuantity) < 0) {
21             throw new IllegalArgumentException("No hi ha prou unitats. La
22                 quantitat en estoc és: " + medicamentById.getStockQuantity());
23         }
24     }
25 }
```

L'anotació `@Autowired` fa que Spring us retorni un objecte sense necessitat de crear-lo directament al nostre codi. En aquest cas voleu fer servir `MedicamentRepository`.

El mètode `processMovimentStock` fa el conjunt d'operacions que requeríem. Demana al repositori el medicament i fa l'actualització del seu estoc amb `setStockQuantity`. Però en cap cas deixarà l'estoc en negatiu, perquè abans llançaria l'excepció `IllegalArgumentException`. Recordeu que al mètode `getMedicamentById` també es lanza una excepció en cas de no existir el medicament.

També us interessa que Spring cerqui automàticament les classes d'aquest paquet per poder injectar els objectes quan es trobi una anotació d'aquest tipus. Per això, canvieu la propietat `context:component-scan` de `DispatcherServlet-servlet.xml`:

```

1  <context:component-scan base-package="cat.xtec.ioc.controller cat.xtec.ioc.
   domain.repository cat.xtec.ioc.service" />
```

2.3.3 Afegeix a la capa de presentació

En comptes de fer servir `MedicamentController` emprareu un nou controlador específic per a aquest cas.

Al paquet `cat.xtec.ioc.controller`, creeu el controlador `MovimentStockController` amb el codi següent:

```

1 package cat.xtec.ioc.controller;
2
3 import cat.xtec.ioc.service.MovimentStockService;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.stereotype.Controller;
6 import org.springframework.web.bind.annotation.RequestMapping;
7
8 @Controller
9 public class MovimentStockController {
```

```

10
11     @Autowired
12     private MovimentStockService movimentStockService;
13
14     @RequestMapping("/movimentestoc/M020/2/-1")
15     public String process() {
16         movimentStockService.processMovimentStock("M020", 2, -1);
17         return "redirect:/medicaments";
18     }
19 }
```

Amb l'URL de `@RequestMapping`, el controlador crida la capa de servei per fer el moviment d'estoc. Si hi ha cap excepció es mostrarà el missatge que toca, però si no mostrarà la llista de medicaments on podeu veure el decrement de l'estoc (signe `-1` en l'exemple).

En aquest cas, l'URL és fix i es crida a fer el moviment amb constants. No us preocupeu, ja veureu com fer tot això variable.

Fins ara havíeu fet servir la creació explícita del `ModelAndView` com a resposta, però també es pot retornar un `string` amb el nom de la vista i Spring crearà l'objecte `ModelAndView` per a vosaltres.

En el cas que us ocupa heu d'anar a la vista `Medicaments`, i per evitar tornar a fer el moviment d'estoc, si l'usuari prem el botó *Anar enrere* del navegador, feu una redirecció (`return "redirect:/medicaments"`). I per fer servir `redirect` heu emprat el tipus `string` com a retorn.

Executeu l'aplicació i aneu a `/medicaments`. El navegador us mostra la llista de medicaments amb l'estoc actual (vegeu la figura 2.7).

FIGURA 2.7. Sortida de l'aplicació Estoc de medicaments, estoc actual

The screenshot shows a web page titled "Medicaments" with a blue header bar containing the text "Lista de medicaments en magatzem". Below the header, there are three cards representing different medications:

- Ibuprofèn: Ibuprofèn 600 mg, 2,0 €, 214 unitats en magatzem
- Paracetamol: Paracetamol 1 g, 2,6 €, 56 unitats en magatzem
- Àcid acetilsalicílic: Àcid acetilsalicílic 2,6 €, 15 unitats en magatzem

Ara canvieu l'URL a `/movimentestoc/M020/2/-1` i veureu el següent resultat, que mostra com ha variat l'estoc del paracetamol (vegeu la figura 2.8).

FIGURA 2.8. Sortida de l'aplicació Estoc de medicaments, estoc modificat

The screenshot shows a web page titled "Medicaments" with a blue header bar containing the text "Lista de medicaments en magatzem". Below the header, there are three cards representing different medications:

- Ibuprofèn: Ibuprofèn 600 mg, 2,0 €, 214 unitats en magatzem
- Paracetamol: Paracetamol 1 g, 2,6 €, 54 unitats en magatzem
- Àcid acetilsalicílic: Àcid acetilsalicílic 2,6 €, 15 unitats en magatzem

The Paracetamol card shows a reduced quantity of 54 units, indicating it has been partially moved.

2.3.4 Què heu fet i què heu après amb la capa de servei?

A l'aplicació Estoc de medicaments heu implementat un moviment d'estoc d'un medicament, comprovant que existeix i que no deixeu l'estoc en negatiu.

Per fer això heu seguit les recomanacions d'Spring MVC i heu creat la capa de **servei**, on s'implementen regles de negoci, és a dir, operacions complexes que poden abastar una o més entitats i que poden contenir operacions simples.

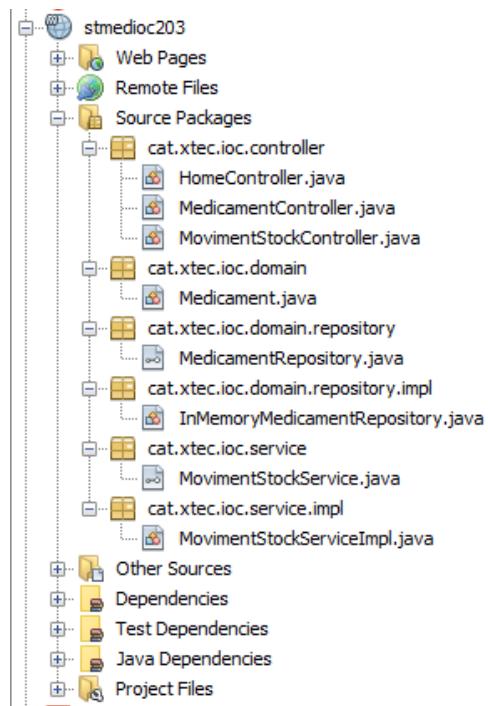
Heu creat un nou controlador que crida la capa de servei, i és aquesta capa la que crida la de persistència. A la vegada, heu afegit els mètodes necessaris a la persistència per complir amb les noves peticions. En el vostre cas, la persistència us proporciona un medicament mitjançant el codi.

2.4 Què s'ha après?

A Estoc de medicaments partíeu d'una pàgina de benvinguda, i ara podeu mostrar la llista de medicaments i podeu fer moviments d'estoc (encara de manera fixa).

Per fer això heu estructurat el projecte en capes, tal com es mostra en la figura 2.9.

FIGURA 2.9. Estructura de capes d'una aplicació Spring MVC



Heu seguit l'estructuració de codi recomanada per a aplicacions web amb Spring MVC amb les capes següents:

- presentació (*Presentation*)

- domini (*Domain*)
- servei (*Service*)
- persistència (*Persistence*)

Heu vist que la capa de **presentació** la conformen les vistes jsp i les classes controladores. A més, mitjançant fitxers de configuració i anotacions, Spring MVC afegeix altres classes necessàries, com el Dispatcher Servlet i View Resolvers.

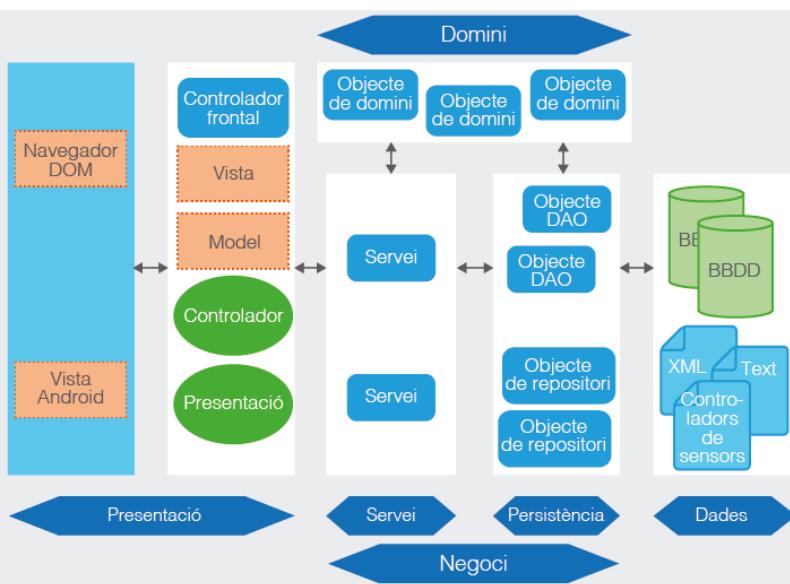
Heu desenvolupat la **persistència** amb objectes repositori que dialoguen amb les dades. En el vostre cas no accediu a una base de dades, perquè no és dins de l'abast d'aquesta unitat; treballau amb dades creades en memòria.

Heu creat la capa **domini** com la representació del nostre model de dades, una classe per a cada entitat. En el vostre cas, una única classe Medicament.

Heu acabat amb la capa de **servei**, que implementa les regles de negoci. En el vostre cas, un moviment d'estoc que es compon de diverses operacions: comprovació de l'existència del medicament, verificar que l'estoc no queda en negatiu i finalment, actualitzar les unitats en estoc.

Heu connectat tot segons l'arquitectura que proposa Spring MVC (vegeu la figura 2.10).

FIGURA 2.10. Capes i interaccions a Spring MVC



Heu fet injecció de dependències i l'heu fet mitjançant interfícies per aconseguir un acoblament feble entre les capes que permet un millor manteniment i adaptació futurs.

L'aplicació Estoc de medicaments us mostra una llista de medicaments, i podeu fer un moviment d'estoc. Però us animo a continuar amb aquests materials per descobrir com fer que el moviment d'estoc sigui variable i moltes coses més, com crear productes amb formularis des del navegador.

3. Spring MVC, aprofundint en els controladors

Partiu de l'aplicació web **Estoc de medicaments** que mostra una pàgina de benvinguda, i que mitjançant l'URL adient pot mostrar una llista de medicaments i fins i tot rebaixar l'estoc d'un d'aquests.

Aquesta aplicació segueix l'arquitectura i el procés de qualsevol aplicació Spring MVC. L'aplicació s'estructura en les quatre capes següents:

- presentació (*Presentation*)
- domini (*Domain*)
- servei (*Service*)
- persistència (*Persistence*)

Estoc de medicaments fa servir anotacions de Spring MVC:

- `@Controller`, `@Repository` i `@Service`, que defineixen el tipus d'objecte Spring.
- `@RequestMapping`, que caça l'URL definit i executa el mètode que el segueix.
- `@Autowired`, que injecta l'objecte que segueix.

A més, Estoc de medicaments conté els fitxers XML amb la configuració necessària per dirigir el comportament de Spring MVC respecte a l'aplicació.

Anem a afegir més funcionalitat a l'aplicació Estoc de medicaments per conèixer els conceptes i procediments associats, aprofundint un punt més en l'estudi dels controladors.

Estoc de medicaments mostra la llista dels medicaments. El procés per mostrar la llista el canviareu per adaptar-lo a un veritable procés Spring MVC, és a dir, creant el servei corresponent i fent que el controlador cridi el servei en comptes del que fa ara, cridar directament la persistència. Aprofitant aquesta implementació, usareu l'anotació `@RequestMapping` a nivell de classe i a nivell de mètode.

Fareu que Estoc de medicaments mostri medicaments d'una categoria, però en aquest cas introduireu com l'URL de `@RequestMapping` pot ser variable (*path variable*) afegint l'anotació `@PathVariable`. Ja us convé, atès que ara, pel moviment d'estoc, l'URL és fix.

Aprofundint en variables a l'URL permetreu que Estoc de medicaments mostri medicaments basats en un **filtre**, és a dir, en un conjunt de parells (propietat,

valor) que implementareu amb l'anotació `@MatrixVariable` i una col·lecció `Map` (*matrix variable*).

Finalment, l'aplicació podrà mostrar el detall d'un medicament. En aquest cas no fareu servir variables a l'URL, sinó que utilitzareu l'anotació `@RequestParam` per definir paràmetres HTTP a partir de peticions `Get` o `Post`.

3.1 Estoc de medicaments, capa de servei a medicament

En l'aplicació Estoc de medicaments de la qual partiu, el controlador `MedicamentController` crida directament la capa de persistència; concretament, es fa un injecció de `MedicamentRepository`.

Arregleu això implementant la capa de servei corresponent i aprofiteu per endreçar una mica les crides fent que mostrar tots els medicaments sigui un cas particular de mostrar medicaments.

El codi del projecte “stmedioc” en l'estat capa de servei a medicament es pot descarregar des de l'enllaç que trobareu als annexos de la unitat. Però per seguir el desenvolupament que segueix és millor partir del que ja heu fet servir i copiar-lo amb el nom de “stmedioc301”.

3.1.1 Creant el servei a medicament

Al paquet `cat.xtec.ioc.service` heu de crear la interfície `MedicamentService` amb el codi que es mostra a continuació:

```

1 package cat.xtec.ioc.service;
2
3 import cat.xtec.ioc.domain.Medicament;
4 import java.util.List;
5
6 public interface MedicamentService {
7
8     List<Medicament> getAllMedicaments();
9
10    Medicament getMedicamentById(String medicamentID);
11}
```

L'objectiu és mostrar la llista de medicaments amb les crides ortodoxes (presentació a servei i aquest a persistència), i per això definiu `getAllMedicaments`. També aprofiteu per endreçar i afegiu el mètode `getMedicamentById`, que fareu servir quan vulgueu un medicament a partir del seu codi.

Al paquet `cat.xtec.ioc.service.impl` heu de crear la classe `MedicamentServiceImpl` que implementi la interfície anterior amb el codi que es mostra.

```

1 package cat.xtec.ioc.service.impl;
2
3 import cat.xtec.ioc.domain.Medicament;
4 import cat.xtec.ioc.domain.repository.MedicamentRepository;
5 import cat.xtec.ioc.service.MedicamentService;
6 import java.util.List;
7 import org.springframework.beans.factory.annotation.Autowired;
```

```

8 import org.springframework.stereotype.Service;
9
10 @Service
11 public class MedicamentServiceImpl implements MedicamentService {
12
13     @Autowired
14     private MedicamentRepository medicamentRepository;
15
16     public List<Medicament> getAllMedicaments() {
17         return medicamentRepository.getAllMedicaments();
18     }
19
20     public Medicament getMedicamentById(String medicamentID) {
21         return medicamentRepository.getMedicamentById(medicamentID);
22     }
23 }
```

En aquest cas, el servei no afegeix més regles de negoci, simplement es limita a cridar la persistència per obtenir el resultat. No obstant això, s'ha d'implementar aquesta capa perquè en el futur és possible que sorgeixin necessitats a implementar en aquesta capa. Per exemple, podríem dir que encara que obtingueu un medicament el retornareu només si l'usuari està autoritzat a la seva categoria.

3.1.2 Ordenant les crides des del controlador

Canviu tot el controlador MedicamentController amb el codi que es mostra a continuació:

```

1 package cat.xtec.ioc.controller;
2
3 import cat.xtec.ioc.service.MedicamentService;
4 import java.io.IOException;
5 import javax.servlet.ServletException;
6 import javax.servlet.http.HttpServletRequest;
7 import javax.servlet.http.HttpServletResponse;
8 import org.springframework.beans.factory.annotation.Autowired;
9 import org.springframework.stereotype.Controller;
10 import org.springframework.web.bind.annotation.RequestMapping;
11 import org.springframework.web.servlet.ModelAndView;
12
13 @Controller
14 @RequestMapping("/medicaments")
15 public class MedicamentController {
16
17     @Autowired
18     private MedicamentService medicamentService;
19
20     @RequestMapping("/all")
21     public ModelAndView allMedicaments(HttpServletRequest request,
22             HttpServletResponse response)
23             throws ServletException, IOException {
24         ModelAndView modelview = new ModelAndView("medicaments");
25         modelview.getModelMap().addAttribute("medicaments", medicamentService.
26             getAllMedicaments());
27         return modelview;
28     }
29 }
```

Heu canviat la injecció del *repository* per la del nou servei. Ara sou a prop de l'ortodòxia a Spring MVC.

L'anotació `@RequestMapping` permet al Servlet Dispatcher encaminar la petició sobre el controlador específic que ha de processar-la.

Ara teniu dos `@RequestMapping`: un a nivell de classe amb l'URL `/medicaments` i un altre a nivell del mètode `allMedicaments` amb l'URL `/all`.

Si convenim que l'URL inicial de l'aplicació és `urlbase` (p. ex.: `localhost/stmedioc301`), quan el DispatcherServlet trobi l'URL `urlbase/medicaments/all`, amb `/medicaments` determinarà que el controlador que ha de gestionar la petició és `MedicamentController`, i amb `/all` el mètode a executar serà `allMedicaments`.

D'aquesta manera, dins de `MedicamentController` podríeu afegir més mètodes que agafin el control quan es doni una petició `urlbase/medicaments/xxx`, on `xxx` és l'URL que posaríeu a l'anotació `@RequestMapping` a nivell del mètode corresponent.

Proveu a executar l'aplicació afegint `/medicaments/all` a l'URL, i us ha de mostrar la llista de medicaments.

3.2 Estoc de medicaments, medicaments per malaltia

Voleu que l'aplicació Estoc de medicaments mostri la llista de medicaments d'una categoria. Pel que sabeu, podeu fer els següents passos d'implementació:

- Un mètode del repositori que doni aquesta llista de medicaments a partir de la categoria donada com a paràmetre.
- Un mètode al servei que cridi aquest mètode del repositori.
- Un mètode al controlador que caci l'URL de la categoria i cridi el servei.

Però amb això, l'anotació `@RequestMapping` al controlador seria fixa amb la categoria com a constant, com per exemple `.../medicaments/Analgèsic`. Amb dues categories, com és el cas de la vostra aplicació, encara es pot suportar, però si en teniu algunes més ja no tindria sentit, ja que hauríeu de posar un `@RequestMapping` amb cada categoria com a URL. Encara més, no sabeu quines categories es poden crear en el futur, i per en cada una hauríeu de modificar el codi.

No us heu de preocupar, perquè podeu fer servir la funcionalitat de Spring MVC anomenada **plantilla d'URI** (*URI template pattern*).

En el cas d'Estoc de medicaments, teniu:

- `localhost:8080/stmedioc301/medicaments/Analgèsic`
- `localhost:8080/stmedioc301/medicaments/Anti-inflamatori`

on podríeu dir que la darrera part de l'URL és variable.

A Spring MVC, aquesta variable es nota entre claus i és una variable de ruta (*path variable*). Si l'anomeneu categoria, llavors l'URL la podríeu considerar com `localhost:8080/stmedioc301/medicaments/{categoria}`.

Amb aquesta definició ja podeu començar a implementar la solució per mostrar els **medicaments per malaltia** amb una variable de ruta.

3.2.1 Adaptant repositori i servei

A la interfície MedicamentRepository, afegiu la següent declaració de mètode:

```
1 List<Medicament> getMedicamentsByCategory(String category);
```

I a la classe InMemoryMedicamentRepository, implementeu el mètode amb el codi que es mostra:

```
1 public List<Medicament> getMedicamentsByCategory(String category) {
2     List<Medicament> medicamentsByCategory = new ArrayList<Medicament>();
3     for (Medicament medicament : listOfMedicaments) {
4         if (category.equalsIgnoreCase(medicament.getCategory())) {
5             medicamentsByCategory.add(medicament);
6         }
7     }
8     return medicamentsByCategory;
9 }
```

El mètode `getMedicamentsByCategory` retornarà una `List` amb objectes Medicament de la categoria passada per paràmetre. Fixeu-vos que ignora diferències entre majúscules i minúscules.

A la interfície MedicamentService, afegiu la següent declaració de mètode:

```
1 List<Medicament> getMedicamentsByCategory(String category);
```

I a la classe MedicamentServiceImpl, implementeu el mètode amb el codi que es mostra:

```
1 public List<Medicament> getMedicamentsByCategory(String category) {
2     return medicamentRepository.getMedicamentsByCategory(category);
3 }
```

Simplement, es crida el repositori sense més regles de negoci a afegir.

El codi del projecte "stmedioc" en l'estat **medicaments per malaltia** es pot descarregar des de l'enllaç que trobareu als annexos de la unitat. Però per seguir el desenvolupament és millor partir del que ja heu fet servir i copiar-lo amb el nom de "stmedioc301".

3.2.2 Adaptant el controlador

A MedicamentController, afegireu l'anotació `@RequestMapping` per caçar qualsevol categoria, és a dir, fareu servir una variable de ruta (*path variable*). I al

mètode que segueix, recuperareu aquesta variable per passar-la com a paràmetre en la cerca de medicaments per categoria.

A la classe MedicamentController, afegiu l'anotació i el mètode amb el codi que es mostra:

```

1  @RequestMapping("/{category}")
2      public ModelAndView getMedicamentsByCategory(@PathVariable("category")
3          String medicamentCategory, HttpServletRequest request,
4          HttpServletResponse response)
5              throws ServletException, IOException {
6      ModelAndView modelview = new ModelAndView("medicaments");
7      modelview.getModelMap().addAttribute("medicaments", medicamentService.
8          getMedicamentsByCategory(medicamentCategory));
9      return modelview;
10 }
```

Us demanarà afegir l'import següent:

```
1 import org.springframework.web.bind.annotation.PathVariable;
```

A @RequestMapping feu servir l'anotació {pathVariable} per indicar a Spring que l'URL contindrà un valor variable en aquesta part. En el vostre cas, el nom de la variable de ruta és *category*.

El mètode que es llança és getMedicamentsByCategory, i en els seus paràmetres heu fet servir l'anotació @PathVariable amb el format:

```
1 @PathVariable(pathVariable) type variableMetode
```

@PathVariable sense valors de paràmetre implica que pathVariable té el mateix nom que variableMetode.

on pathVariable és el nom de la variable de ruta definida a l'anotació @RequestMapping, variableMetode és la variable que fareu servir al mètode, i type és el seu tipus. Així, Spring guarda el valor que s'ha posat a l'URL a variableMetode i ja la podeu fer servir amb aquest nom en el codi inclòs en el mètode.

En el vostre cas, si executeu l'aplicació per exemple amb l'URL *localhost:8080/stmedioc302/medicaments/Analgesic*, el DispatcherServlet, com que l'URL és *urlbase/medicaments/...*, passarà el control a MedicamentController. Dins d'aquest controlador es crida el mètode precedit per l'anotació @RequestMapping(''{category}''), ja que s'ha passat un URL del tipus *urlbase/medicaments/xxx*, on *xxx* és qualsevol valor diferent d'*all* (existeix @RequestMapping(''{all}''), que cridaria allMedicaments).

El mètode getMedicamentsByCategory cridat demana al repositori els medicaments de la categoria que s'ha passat per paràmetre i mostra la vista de medicaments només amb els que ha trobat.

Així, el resultat per a l'URL passat hauria de ser el que es mostra en la figura 3.1.

FIGURA 3.1. Sortida de medicaments per categoria

Medicaments	
Llista de medicaments en magatzem	
Paracetamol Paracetamol 1 g 2,6 € 56 unitats en magatzem	Àcid acetilsalicílic Àcid acetilsalicílic 2,6 € 15 unitats en magatzem

3.3 "Estoc de medicaments", llista de medicaments segons filtre

“Estoc de medicaments” pot mostrar medicaments d’una categoria passant l’URL amb `/medicaments/nomDeCategoria`, on `nomDeCategoria` serà rebut al controlador específic amb l’anotació `@RequestMapping(''/{}{category}'')` que conté la variable de ruta (*path variable*).

Els vostres requeriments es veuen ampliats amb la necessitat de mostrar medicaments segons filters. Enriquiu l’aplicació acceptant URL del tipus `localhost:8080/stmedioc303/medicaments/filter/ByCriteria;producer=Ferrer,Bayer;estoc=1,100`, que, per exemple, podeu traduir com a llista de medicaments dels productors Ferrer o Bayer amb unitats en estoc entre 1 i 100. Però com veureu, la traducció és un conveni, perquè depèn del que implementeu a la consulta que fareu a les dades.

En realitat, el que voleu implementar és l’acceptació de variables de matriu (*matrix variables*) com a paràmetres i com fer-les servir al vostre codi.

Aquestes variables es convertiran en un `Map<String, List<String>>` a partir de l’anotació `@MatrixVariable`. Fixeu-vos que té sentit, perquè `producer=Ferrer,Bayer;estoc=1,100` es pot representar com una `Map` amb les *keys* `producer` i `estoc`, i els valors són llistes.

Implementeu l’exemple que dóna resposta a `localhost:8080/stmedioc303 /medicaments/filter/ByCriteria;producer=Ferrer,Bayer;estoc=1,100` amb el significat que hem convingut.

3.3.1 Adaptant repositori i servei

A la interfície `MedicamentRepository`, afegiu la següent declaració de mètode:

```
1 Set<Medicament> getMedicamentsByFilter(Map<String, List<String>> filterParams);
```

I a la classe `InMemoryMedicamentRepository`, implementeu el mètode amb el codi que es mostra.

```
1 public Set<Medicament> getMedicamentsByFilter(Map<String, List<String>>
    filterParams) {
```

El codi del projecte “stmedioc” en l'estat **llista de medicaments segons filtre** es pot descarregar des de l'enllaç que trobareu als annexos de la unitat. Però per seguir el desenvolupament és millor partir del que ja heu fet servir i copiar-lo amb el nom de “stmedioc303”.

```

2     Set<Medicament> medicamentsByProducer = new HashSet<Medicament>();
3     Set<Medicament> medicamentsInStockRange = new HashSet<Medicament>();
4     Set<String> criterias = filterParams.keySet();
5     long minStock = 0;
6     long maxStock = 0;
7     if (criterias.contains("producer")) {
8         for (String producerName : filterParams.get("producer")) {
9             for (Medicament medicament : listOfMedicaments) {
10                 if (producerName.equalsIgnoreCase(medicament.getProducer()))
11                     )
12                     medicamentsByProducer.add(medicament);
13                 }
14             }
15         }
16         if (criterias.contains("estoc")) {
17             minStock = Long.parseLong(filterParams.get("estoc").get(0));
18             maxStock = Long.parseLong(filterParams.get("estoc").get(1));
19
20             for (Medicament medicament : listOfMedicaments) {
21                 if ((medicament.getStockQuantity() >= minStock) && (medicament.
22                     getStockQuantity() <= maxStock))
23                     medicamentsInStockRange.add(medicament);
24                 }
25             }
26         }
27         medicamentsInStockRange.retainAll(medicamentsByProducer);
28     return medicamentsInStockRange;
29 }
```

Set és una interfície del framework Java Collections.

Es pot consultar la referència a la mateixa documentació d'Oracle a bit.ly/2r3n02Y.

Es defineixen dues col·leccions (`medicamentsByProducer` i `medicamentsInStockRange`) que contindran els medicaments que compleixen els criteris per retornar només una amb la intersecció d'elements.

És defineix un *Set* de criteris que en el vostre cas contindrà *producer* i *estoc*. Fixeu-vos que podreu provar sense algun dels criteris, i llavors aquest no es tindrà en compte.

En el cas del criteri *producer*, se seleccionen els elements del paràmetre amb aquest criteri, es recorre aquesta selecció i per a cada valor s'afegeixen els medicaments amb aquest *producer* al *Set* `medicamentsByProducer`.

En el cas del criteri *estoc*, es fan servir les variables *minStock* i *maxStock* per assignar respectivament els elements 0 i 1 de la llista de paràmetres amb la *key* *estoc* (`filterParams.get("estoc").get(i)`). Llavors, es van afegint sobre el *Set* `medicamentsInStockRange` els medicaments amb unitats en estoc en el rang $[minStock, maxStock]$.

Finalment, com que voleu retornar la llista de medicaments que compleixen tots els criteris a la vegada, agafeu una de les llistes i només la deixeu amb els medicaments que coincideixen amb l'altre, per finalment retornar la intersecció.

```
1 medicamentsInStockRange.retainAll(medicamentsByProducer);
```

Un cop heu definit la consulta a les dades, passeu a modificar la capa de servei.

A la interfície `MedicamentService`, afegiu la següent declaració de mètode:

```
1 Set<Medicament> getMedicamentsByFilter(Map<String, List<String>> filterParams);
```

I a la classe MedicamentServiceImpl, implementeu el mètode amb el codi que es mostra:

```
1 public Set<Medicament> getMedicamentsByFilter(Map<String, List<String>>
2     filterParams) {
3     return medicamentRepository.getMedicamentsByFilter(filterParams);
}
```

Simplement es crida el repositori, sense més regles de negoci a afegir.

3.3.2 Adaptant el controlador

Al controlador específic voleu fer servir variables tipus matriu, com per exemple les que es mostren a l'URL `localhost:8080 /stmedic303/medicaments/filter/ByCriteria;producer=Ferrer,Bayer;estoc=1,100`.

Tal com heu implementat al mètode getMedicamentsByFilter del repositori, heu convingut que això voldrà dir el conjunt de medicaments amb productor Bayer o Ferrer, i a més l'estoc dels quals estigui entre 1 i 100 unitats.

Per aconseguir això es farà servir l'anotació `@MatrixVariable`, però, a més, a Spring MVC es necessita una configuració addicional.

A DispatcherServlet-servlet.xml canvieu l'etiqueta `mvc:annotation-driven` així:

```
1 <mvc:annotation-driven enable-matrix-variables="true"/>
```

A la classe MedicamentController, afegiu l'anotació i el mètode amb el codi que es mostra:

```
1 @RequestMapping("/filter/{ByCriteria}")
2     public ModelAndView getMedicamentsByFilter(@MatrixVariable(pathVar =
3         "ByCriteria") Map<String, List<String>> filterParams,
4         HttpServletRequest request, HttpServletResponse response)
5             throws ServletException, IOException {
6     ModelAndView modelview = new ModelAndView("medicaments");
7     modelview.getModelMap().addAttribute("medicaments",
8         medicamentService.getMedicamentsByFilter(filterParams));
9     return modelview;
10 }
```

A `@RequestMapping` feu servir l'anotació `{pathVariable}` per indicar a Spring que l'URL contindrà un valor variable en aquesta part. En el vostre cas, el nom de la variable és `ByCriteria`.

El mètode que es llança és `getMedicamentsByFilter`, i en els seus paràmetres hem fet servir l'anotació `@MatrixVariable` amb el format:

```
1 @MatrixVariable(pathVar = "pathVariable") Map<String, List<String>>
2     filterParams
```

L'anotació `@RequestMapping` permet definir més d'un grup de criteris, és a dir, més d'un grup de *matrix variables*. El format seria així: `@RequestMapping("/filter/{ByCriteria}/{ByOtherCriteria})`.

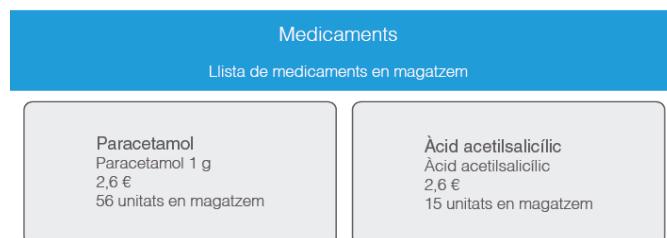
on `pathVariable` és el nom de la variable definida a l'anotació `@RequestMapping` i `filterParams` és la variable que farem servir al mètode amb tipus *Map*. Així, Spring guarda el valor que s'ha posat a l'URL a `filterParams` i ja la podeu fer servir amb aquest nom en el codi inclòs en el mètode.

En el vostre cas, si executeu l'aplicació per exemple amb l'URL *localhost:8080/stmedioc303/medicaments/filter/ByCriteria;producer=Ferrer,Bayer;estoc=1,100*, com que l'URL és *urlbase/medicaments/...*, el `DispatcherServlet` passarà el control a `MedicamentController`. Dins d'aquest controlador es crida el mètode precedit per l'anotació `@RequestMapping("/filter/{ByCriteria})`, ja que s'ha passat un URL del tipus *urlbase/medicaments/filter/ByCriteria;xxx*, on *xxx* és un conjunt de parells clau=llista de valors.

El mètode `getMedicamentsByFilter` cridat demana al repositori els medicaments amb el filtre que s'ha passat per paràmetre i mostra la vista medicaments només amb els que ha trobat. Fixeu-vos que el paràmetre “filtre”, conjunt de parells clau=llista de valors, s'ha convertit a un *Map<String, List<String>>* que anomenem `filterParams`, i amb aquest nom el fem servir al codi del mètode.

Així, el resultat per a l'URL passat hauria de ser el que es mostra en la figura 3.2.

FIGURA 3.2. Sortida de la llista de medicaments amb filtre



Podeu provar URL amb altres rangs d'estoc i altres productors.

3.4 "Estoc de medicaments", detall de medicament

“Estoc de medicaments” mostra llistes de medicaments que compleixen certs criteris basats en variables que posem a l'URL. És a dir, en els casos que es gestionen amb les anotacions `@PathVariable` i `@MatrixVariable`, s'estan rebent els valors com a part de l'URL i no com a variables que formen part d'*HTTP request*.

Amplieu l'aplicació per tal d'acceptar variables a la petició *HTTP* (*HTTP request*). Per exemple, analitzant l'URL *localhost:8080/stmedioc304/medicaments/medicament?codi=M020* veieu que `codi` serà passat al servidor dins del cos de la petició *HTTP* com una variable.

Aquests tipus d'URL tenen el format *urlbase/...?var1=valor1&var2=valor2&...&varn=valorn*, on el símbol “?” indica que comença la llista de variables

i el símbol & separa cada variable. Cada parell variable=valor serà passat al servidor web dins del cos de l'*HTTP request* (petició *HTTP*).

Per exemple, encara que doni un error, si obriu el navegador i Firebug i provem l'URL `localhost/?var1=valor1&var2=valor2&var3=valor3`, veureu que heu fet un get i ha passat per paràmetre:

- var1 valor1
- var2 valor2
- var3 valor3

I aquest fet no es dóna quan passeu les URL del tipus que gestionen `@PathVariable` i `@MatrixVariable`, ja que no es correspon amb la sintaxi `urlbase/...?var1=valor1&var2=valor2&...&varn=valorn` i, per tant, no hi ha paràmetres al cos de la petició *HTTP*.

A “Estoc de medicaments”, per il·lustrar el pas de variables a la petició *HTTP*, veureu com mostrar el detall d'un medicament a partir d'un URL del tipus esmentat, és a dir, passant els valors com a variables del cos de l'*HTTP request*. Per fer això fareu servir l'anotació `@RequestParam`.

No només veureu això, també fareu que des de qualsevol llista de medicaments, filtrada o no, es pugui anar al detall de qualsevol d'ells amb un botó, i així fareu servir el que heu preparat per mostrar el detall.

3.4.1 Mostrant el detall

Heu de mostrar el detall de medicament en una nova vista que anomenareu `medicament.jsp`. Cal aquesta vista, perquè a les llistes no es mostren totes les dades del medicament, com per exemple la categoria.

Aquesta nova vista es mostrerà quan es doni una petició del tipus `localhost:8080/stmedioc304/medicaments/medicament?codi=M020`.

I per tant, a `MedicamentController` haureu d'afegir l'anotació que agafi el control i el mètode a disparar.

A les diferents capes d'Estoc de medicaments ja teniu la possibilitat d'obtenir un objecte `Medicament` a partir del seu codi, i per això us estalviu codificar en altres capes, a banda de la de presentació.

A la classe `MedicamentController`, afegiu l'anotació i mètode amb el codi que es mostra;

El codi del projecte “stmedioc” en l'estat detall de medicament es pot descarregar des de l'enllaç que trobareu als annexos de la unitat. Però per seguir el desenvolupament és millor partir del que ja heu fet servir i copiar-lo amb el nom de “stmedioc304”.

```

1 @RequestMapping("/medicament")
2     public ModelAndView getMedicamentById(@RequestParam("codi") String
        medicamentId, HttpServletRequest request, HttpServletResponse response
    )

```

```

3           throws ServletException, IOException {
4       ModelAndView modelview = new ModelAndView("medicament");
5       modelview.getModelMap().addAttribute("medicament", medicamentService.
6           getMedicamentById(medicamentId));
7       return modelview;
}

```

L'anotació `@RequestParam` permet assignar la variable passada per paràmetre des de la petició, amb la variable que es farà servir en el mètode.

En el nostre cas, l'anotació `@RequestParam` enllaça el paràmetre codi del cos de la petició amb la variable `medicamentId` que fareu servir dins del mètode.

El mètode crida `getMedicamentById` de la capa de servei que ja està implementat. Aquest retorna l'objecte `Medicament` amb el codi que s'ha passat pel paràmetre.

A la carpeta `views`, afegiu la nova vista `medicament.jsp` amb el codi que es mostra a continuació:

```

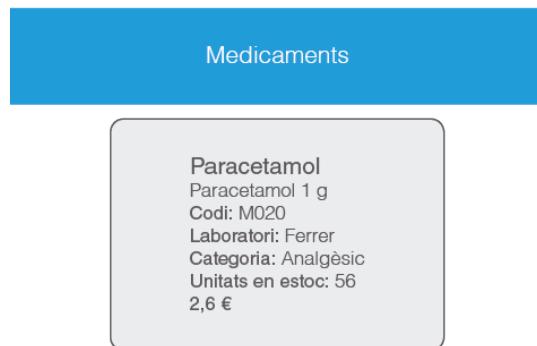
1 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
2 <%@page contentType="text/html" pageEncoding="UTF-8"%>
3 <!DOCTYPE html>
4 <html>
5     <head>
6         <meta http-equiv="Content-Type" content="text/html; charset=ISO
7             -8859-1">
8         <link rel="stylesheet" href="//netdna.bootstrapcdncdn.com/bootstrap/3.0.0/
9             css/bootstrap.min.css">
10        <title>Medicament</title>
11    </head>
12    <body>
13        <section>
14            <div class="jumbotron">
15                <div class="container">
16                    <h1>Medicament</h1>
17                </div>
18            </section>
19            <section class="container">
20                <div class="row">
21                    <div class="col-md-5">
22                        <h3>${medicament.name}</h3>
23                        <p>${medicament.description}</p>
24                        <p>
25                            <strong>Codi : </strong>${medicament.medicamentId}
26                        </p>
27                        <p>
28                            <strong>Laboratori</strong> : ${medicament.producer}
29                        </p>
30                        <p>
31                            <strong>Categoria</strong> : ${medicament.category}
32                        </p>
33                        <p>
34                            <strong>Unitats en estoc </strong> :
35                            ${medicament.stockQuantity}
36                        </p>
37                        <h4>${medicament.price} USD</h4>
38                    </div>
39                </div>
40            </section>
41        </body>
42    </html>

```

La vista mostrarà els valors de l'objecte de nom medicament que havíeu afegit com a atribut al controlador.

Si executeu l'aplicació al navegador amb l'URL `localhost:8080/stmedioc304/medicaments/medicament?codi=M020`, el resultat és el que es veu en la figura 3.3.

FIGURA 3.3. Sortida de detall de medicament



La part de l'URL `/medicaments` ha fet que el Dispatcher Servlet passi la responsabilitat a MedicamentController. L'URL és `/medicaments/medicament`, i com que existeix l'anotació `@RequestMapping("/medicament")` dispararà el mètode `getMedicamentById` que la segueix.

Aquest mètode té anotacions del tipus `@RequestParam`, i per tant farà servir els paràmetres del cos de la petició. En concret, l'URL conté `?codi=M020`, que vol dir que caçarà el paràmetre `codi` a partir de `@RequestParam("codi") String medicamentId` i l'assignarà a `medicamentId`, que després es fa servir per cercar l'objecte Medicament amb igual codi. Aquest objecte Medicament és ficat en un atribut `medicament` a la resposta (*response*).

Llavors, la vista `medicament.jsp` mostra les dades d'aquest medicament fent servir l'atribut esmentat. Proveu amb altres codis de medicaments per veure'n el funcionament.

3.4.2 Arribant al detall des de la llista

“Estoc de medicaments” sap com mostrar el detall d'un medicament a partir de paràmetres de petició, és a dir, amb URL del tipus `localhost:8080/stmedioc304/medicaments/medicament?codi=xxx`, on `xxx` és el `medicamentId`.

Aproveieu el que heu fet per mostrar el detall de qualsevol medicament a partir de la llista. Afegireu un botó a cada element de la llista que ens porti a la vista de detall. I per no anar canviant URL, fareu que des del detall es pugui tornar a la llista.

A `medicaments.jsp`, afegiu la directiva:

```
1 <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

i just després de:

```
1 <p>Hi ha ${medicament.stockQuantity} unitats en magatzem</p>
```

afegiu el codi següent:

```
1 <p>
2   <a href="

```

L'enllaç té l'etiqueta `<spring:url>`, que construirà un URL Spring MVC vàlida. En aquest cas, el codi que farà servir l'obté de l'objecte `medicament` de la llista que s'està recorrent.

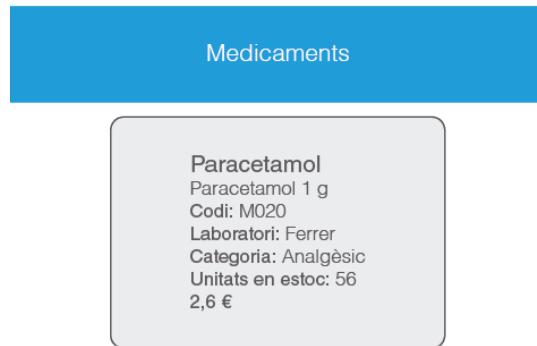
Executeu l'aplicació amb l'URL `localhost:8080/stmedioc304/medicaments/all` i obtindreu un resultat com el que mostra figura 3.4.

FIGURA 3.4. Sortida de llista de medicaments amb botó de detall



Si premeu al detall del Paracetamol, l'aplicació ha de mostrar el detall (vegeu la figura 3.5).

FIGURA 3.5. Sortida de detall de medicament



Ara afegireu al detall un botó per tornar a la llista. Per fer això també es farà servir l'etiqueta `<spring:url>`.

A medicament.jsp, afegiu la directiva:

```
1 <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

i just després de

```
1 <h4>${medicament.price} USD</h4>
```

afegiu el codi següent:

```
1 <a href="

```

Si executeu l'aplicació i aneu a mostrar la llista de medicaments ja podeu premer un botó de detall, que farà que es mostri la vista de detall, i després tornar una altra vegada a la llista.

3.5 Què s'ha après?

MedicamentController crida directament a la capa de servei MedicamentService i no directament a la persistència.

L'anotació `@RequestMapping('/medicaments')` a nivell de classe, fa que qualsevol URL del tipus `/medicaments` agafi aquest controlador. Les anotacions `@RequestMapping('/xxx')` posades al davant de diversos mètodes, fa que aquests es disparin quan es fan peticions amb URL del tipus `/medicaments/xxx`.

Peticions com `localhost:8080/stmedioc301/medicaments/Analgesic`, on “analgesic” es pot substituir per qualsevol categoria, són ateses pel mètode que segueix a l'anotació:

```
1 @RequestMapping("/{category}")
```

on `category` és la variable de ruta (*variable path*) que és enllaçada a la variable `medicamentCategory` amb l'anotació:

```
1 @PathVariable("category") String medicamentCategory.
```

Les variables de matriu (*matrix variables*), permeten llegir URL del tipus `localhost:8080/stmedioc303/medicaments/filter/ByCriteria;producer=Ferrer,Bayer;estoc=1,100`, on el conjunt de parell clau=llista de valors és la matriu.

Amb l'anotació:

```
1 @RequestMapping("/filter/{ByCriteria}")
```

es passa el control al mètode que la segueix, i en aquest, l'anotació:

```
1 @MatrixVariable(pathVar = "ByCriteria") Map<String, List<String>> filterParams
```

assigna el paràmetre (la matriu) al *Map* `filterParams`, variable que es pot fer servir dins del mètode.

Poden venir paràmetres al cos d'una petició HTTP, per exemple, a una petició Post d'un formulari.

En general, es pot atendre a peticions amb URL del tipus *urlbase...?var1=valor1&var2=valor2&...&varn=valorn*, on el que segueix el símbol ? és el conjunt de paràmetres que aniran en el cos de la petició HTTP.

Per exemple, es pot resoldre una URL com:

localhost:8080/stmedioc304/medicaments/medicament?codi=M020

amb l'anotació:

```
1 @RequestMapping("/medicament")
```

precedint al mètode i com a paràmetre d'aquest, l'anotació:

```
1 @RequestParam("codi") String medicamentId
```

que enllaça el paràmetre amb la variable `medicamentId` que es pot fer servir dins del mètode.

L'etiqueta de Spring `<spring:url>` serveix per enllaçar una pàgina amb una altra construint un URL vàlid.

4. Spring MVC, altres aplicacions

Partim de l'aplicació web Estoc de medicaments, que mostra una pàgina de benvinguda i, mitjançant l'URL adient, pot proporcionar la llista de tots els medicaments, els d'una categoria o els medicaments que compleixen cert filtre.

També pot mostrar el detall d'un medicament directament amb un URL o des de la llista de medicaments. Afegit a tot això, l'aplicació pot actualitzar l'estoc d'un medicament.

Aquesta aplicació segueix l'arquitectura i el procés de qualsevol aplicació Spring MVC. S'estructura en les quatre capes següents:

- presentació (*Presentation*)
- domini (*Domain*)
- servei (*Service*)
- persistència (*Persistence*)

Estoc de medicaments fa servir anotacions de Spring MVC:

- `@Controller`, `@Repository` i `@Service`, que defineixen el tipus d'objecte Spring.
- `@RequestMapping`, que caça l'URL definit i executa el mètode que el segueix.
- `@Autowired`, que injecta l'objecte que segueix.
- `@PathVariable` i `@MatrixVariable` per accedir a variables dins de l'URL.
- `@RequestParam` per enllaçar amb els paràmetres del cos de la petició (Request params).

A més, Estoc de medicaments conté els fitxers XML amb la configuració necessària per dirigir el comportament de Spring MVC respecte a l'aplicació.

Anem a afegir funcionalitat a l'aplicació Estoc de medicaments que necessitareu en la majoria d'aplicacions web.

Afegireu a Estoc de medicaments un formulari per crear medicaments, la qual cosa us permetrà veure totes les possibilitats que ofereix Spring MVC per tractar els formularis, sobretot com passar dades des de la vista fins al model, perquè fins ara el sentit havia estat al contrari.

Construireu una pàgina de *login* (identificació) i el codi necessari a la resta de capes. Al voltant d'aquest objectiu, veureu etiquetes Spring relacionades amb la seguretat i diverses possibilitats de configuració.

Fareu servir la **internacionalització** de Spring per mostrar com fer l'aplicació multiidioma (parcialment), i veureu què són i com es comporten els **interceptors** de Spring.

Fareu servir diverses etiquetes pròpies de Spring amb les directives:

```

1 <%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
2 <%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

Aquestes llibreries d'etiquetes proporcionen molta funcionalitat per als formularis, com ara la validació d'errors, però també per a altres finalitats.

4.1 Estoc de medicaments, formulari per afegir medicaments

A Estoc de medicaments, a tota la funcionalitat que hi vegeu implementada, el flux de dades és des del model fins a la vista.

Per veure com es genera i manega a Spring MVC un flux de dades des de la vista fins al model afegireu un formulari l'acció del qual sigui crear un medicament.

Un *form-backing bean* de Spring MVC és l'objecte que permet associar les dades d'un formulari web amb l'aplicació que rep la petició.

L'anotació `@ModelAttribute` i l'objecte *form-backing bean* ens permetran crear el medicament a partir de les dades del formulari.

Els paràgrafs següents descriuen el procés que heu de seguir per implementar el formulari.

Per presentar el formulari implementareu un primer mètode que passarà un objecte Medicament buit (`newMedicament`) com atribut a la nova vista `addMedicament`.

A la vista, l'etiqueta de formulari serà de la llibreria de Spring i associarà bidireccionalment `newMedicament` al formulari, és a dir, cada camp del formulari estarà també associat a les propietats de `newMedicament` i l'actualització d'uns farà que s'actualitzin els altres. Per això es diu que els objectes com `newMedicament` són els *form-backing bean*, ja que és on Spring MVC guarda els valors del formulari en el context.

Finalment, implementareu un segon mètode que reculli el *submit* del formulari. L'anotació:

```
1 @ModelAttribute("newMedicament") Medicament newMedicamentToAdd
```

us permetrà fer servir els valors del formulari via `newMedicamentToAdd`.

El codi del projecte "stmedioc" en l'estat **formulari per afegir medicament** es pot descarregar des de l'enllaç que trobareu als annexos de la unitat.

Però per seguir el desenvolupament és millor partir del que ja heu fet servir i copiar-lo amb el nom de "stmedioc401".

4.1.1 Treballant les capes de domini, persistència i servei

A la classe Medicament, afegiu un constructor sense paràmetres.

```
1 public Medicament() {
2 }
```

És necessari, perquè necessitem crear un objecte medicament buit.

A la interfície MedicamentRepository, afegiu la següent declaració de mètode:

```
1 void addMedicament(Medicament medicament);
```

I a la classe InMemoryMedicamentRepository, implementeu el mètode amb el codi que es mostra.

```
1 public void addMedicament(Medicament medicament) {
2     listOfMedicaments.add(medicament);
3 }
```

Veieu que simplement s'afegeix un objecte Medicament a la vostra llista de medicaments en memòria. Això és equivalent a un INSERT en el cas que la nostra persistència hagués estat implementada amb una base de dades (relacional amb SQL).

A la interfície MedicamentService, afegiu la següent declaració de mètode:

```
1 void addMedicament(Medicament medicament);
```

I a la classe MedicamentServiceImpl, implementeu el mètode amb el codi que es mostra.

```
1 public void addMedicament(Medicament medicament) {
2     medicamentRepository.addMedicament(medicament);
3 }
```

Simplement es crida el repositori sense més regles de negoci a afegir.

4.1.2 Formulari a la capa de presentació

Al controlador heu d'implementar els mètodes inclosos en el procés que havíeu de seguir. És a dir, un mètode per mostrar el formulari i un mètode per processar-lo.

A la classe MedicamentController, heu d'afegir els mètodes amb el codi que es mostra.

```
1 @RequestMapping(value = "/add", method = RequestMethod.GET)
2     public ModelAndView getAddNewMedicamentForm(HttpServletRequest request,
3             HttpServletResponse response)
4             throws ServletException, IOException {
```

```

4      ModelAndView modelAndView = new ModelAndView("addMedicament");
5      Medicament newMedicament = new Medicament();
6      modelAndView.getModelMap().addAttribute("newMedicament", newMedicament);
7      return modelAndView;
8  }
9
10 @RequestMapping(value = "/add", method = RequestMethod.POST)
11 public String processAddNewMedicamentForm(@ModelAttribute("newMedicament")
12     Medicament newMedicamentToAdd) {
13     medicamentService.addMedicament(newMedicamentToAdd);
14     return "redirect:/medicaments/all";
15 }
```

El mètode `getAddNewMedicamentForm` és el que fareu servir per mostrar el formulari. En aquest cas, es dispararà amb una petició GET (`method = RequestMethod.GET`) creant un objecte buit `Medicament` que retorna a la vista `addMedicament` com a atribut de nom `newMedicament`. L'associació que fareu en aquesta vista el converteix en el *form-backing bean*.

El mètode `processAddNewMedicamentForm` és el que fareu servir per processar el formulari. En aquest cas, es dispararà amb una petició POST (`method = RequestMethod.POST`). L'anotació:

```
1 @ModelAttribute("newMedicament") Medicament newMedicament
```

associa les dades del formulari (guardades a `newMedicament`) a la variable `newMedicamentToAdd` que fareu servir al codi del mètode.

Com que no heu de tornar un `ModelAndView` perquè només voleu redirigir a la vista que mostra la llista de medicaments, feu que `processAddNewMedicamentForm` sigui de tipus *string*. No manegueu `ModelAndView` ni necessiteu `HttpServletRequest request` i `HttpServletResponse response` com a paràmetres.

La instrucció:

```
1 return "redirect:/medicaments/all";
```

fa que el navegador faci una nova petició amb aquest URL i, per això, el Dispatcher Servlet tornarà a cercar quin controlador específic s'encarrega de l'URL passat com a *redirect*. En aquest cas és el mateix controlador, i mostrerà la llista de tots els medicaments.

Abans de codificar la vista, a l'arxiu de configuració `web.xml` heu d'afegir el filtre `CharacterEncodingFilter` amb les etiquetes que es mostren.

```

1 <filter>
2   <filter-name>encodingFilter</filter-name>
3   <filter-class>org.springframework.web.filter.CharacterEncodingFilter</
4     filter-class>
5   <init-param>
6     <param-name>encoding</param-name>
7     <param-value>UTF-8</param-value>
8   </init-param>
9   <init-param>
10    <param-name>forceEncoding</param-name>
11    <param-value>true</param-value>
```

```

11      </init-param>
12  </filter>
13  <filter-mapping>
14      <filter-name>encodingFilter</filter-name>
15      <url-pattern>/*</url-pattern>
16  </filter-mapping>
```

Aquest filtre permet la utilització d'accents als camps del nostre formulari. En realitat, `CharacterEncodingFilter` permet l'especificació de la codificació de caràcters que forcem en les peticions, ja que actualment existeixen navegadors que no fan cas de la codificació especificada en la pàgina HTML (a la vista, en el vostre cas).

Afegiu a la carpeta `views` una nova vista `addMedicament.jsp` amb el codi que es mostra a continuació:

```

1 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
2 <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
3 <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
4 <%@page contentType="text/html" pageEncoding="UTF-8"%>
5 <!DOCTYPE html>
6 <html>
7     <head>
8         <meta http-equiv="Content-Type" content="text/html; charset=ISO
9             -8859-1">
10        <link
11            rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/
12                css/bootstrap.min.css">
13        <title>Medicaments</title>
14    </head>
15    <body>
16        <section>
17            <div class="jumbotron">
18                <div class="container">
19                    <h1>Medicament</h1>
20                    <p>Afegir medicament</p>
21                </div>
22            </div>
23        </section>
24        <section class="container">
25            <form:form modelAttribute="newMedicament" class="form-horizontal">
26                <fieldset>
27                    <legend>Afegir medicament</legend>
28                    <div class="form-group">
29                        <label class="control-label col-lg-2 col-lg-2" for=
30                            "medicamentId">Codi</label>
31                        <div class="col-lg-10">
32                            <form:input id="medicamentId" path="medicamentId"
33                                type="text" class="form:input-large"/>
34                        </div>
35                    </div>
36                    <div class="form-group">
37                        <label class="control-label col-lg-2 col-lg-2" for=
38                            "name">Nom</label>
39                        <div class="col-lg-10">
40                            <form:input id="name" path="name" type="text" class
41                                ="form:input-large"/>
42                        </div>
43                    </div>
44                    <div class="form-group">
45                        <label class="control-label col-lg-2 col-lg-2" for=
46                            "price">Preu</label>
47                        <div class="col-lg-10">
48                            <form:input id="price" path="price" type="text"
49                                class="form:input-large"/>
50                        </div>
51                    </div>
52                </fieldset>
53            </form:form>
54        </div>
55    </body>
56 </html>
```

```

43         </div>
44         <div class="form-group">
45             <label class="control-label col-lg-2 col-lg-2" for="producer">Laboratori</label>
46             <div class="col-lg-10">
47                 <form:input id="producer" path="producer" type="text" class="form:input-large"/>
48             </div>
49         </div>
50         <div class="form-group">
51             <label class="control-label col-lg-2 col-lg-2" for="category">Categoria</label>
52             <div class="col-lg-10">
53                 <form:input id="category" path="category" type="text" class="form:input-large"/>
54             </div>
55         </div>
56         <div class="form-group">
57             <label class="control-label col-lg-2 col-lg-2" for="stockQuantity">Unitats en estoc</label>
58             <div class="col-lg-10">
59                 <form:input id="stockQuantity" path="stockQuantity" type="text" class="form:input-large"/>
60             </div>
61         </div>
62         <div class="form-group">
63             <label class="control-label col-lg-2 col-lg-2" for="stockInOrder">Unitats en comandes</label>
64             <div class="col-lg-10">
65                 <form:input id="stockInOrder" path="stockInOrder" type="text" class="form:input-large"/>
66             </div>
67         </div>
68         <div class="form-group">
69             <label class="control-label col-lg-2" for="description">Descripció</label>
70             <div class="col-lg-10">
71                 <form:textarea id="description" path="description" rows="2"/>
72             </div>
73         </div>
74         <div class="form-group">
75             <label class="control-label col-lg-2" for="active">actiu</label>
76             <div class="col-lg-10">
77                 <form:radio button path="active" value="true" />Si
78                 <form:radio button path="active" value="false" />No
79             </div>
80         </div>
81         <div class="form-group">
82             <div class="col-lg-offset-2 col-lg-10">
83                 <input type="submit" id="btnAdd" class="btn btn-primary" value="Crear"/>
84             </div>
85         </div>
86     </fieldset>
87 </form:form>
88 </section>
89 </body>
90 </html>
91
92
93

```

Fixeu-vos que es fan servir les llibreries d'etiquetes de Spring que aporten més funcionalitat.

L'etiqueta que defineix el formulari:

1 <form:form modelAttribute="newMedicament" class="form-horizontal">

està associant els camps del formulari amb el *form-backing bean newMedicament*, de manera que quan s'enviï el formulari, el controlador podrà agafar els valors d'aquest objecte.

Les etiquetes dels camps del formulari:

```
1 <form:input id="medicamentId" path="medicamentId" type="text" class="form:input
-lARGE"/>
```

associen individualment cada control del formulari amb la propietat del *form-backing bean* mitjançant l'atribut *path*.

Si executeu l'aplicació amb l'URL *localhost:8080/stmedioc401/medicaments/add* us mostrerà la pàgina amb el formulari buit (vegeu la figura 4.1).

FIGURA 4.1. Sortida de l'aplicació amb el formulari buit

The screenshot shows a web page titled "Medicament" with a sub-section "Afegeix medicament". Below this, there is a form titled "Afegeix medicament". The form contains the following fields:

Codi	<input type="text"/>
Nom	<input type="text"/>
Preu	<input type="text"/> 0.0
Laboratori	<input type="text"/>
Categoría	<input type="text"/>
Unitats en estoc	<input type="text"/> 0
Unitats en comanda	<input type="text"/> 0
Descripció	<input type="text"/>

Below the form, there is a radio button group for "Actiu": "Sí" (radio button checked) and "No" (radio button unselected). At the bottom of the form is a blue "Crea" button.

Ompliu el formulari tal com es mostra en la figura 4.2.

FIGURA 4.2. Sortida de l'aplicació amb el formulari emplenat

The screenshot shows the same web page and form structure as in Figure 4.1, but with populated fields. The data entered is:

Codi	M230
Nom	Crema
Preu	12
Laboratori	General
Categoría	Cosmètic
Unitats en estoc	12
Unitats en comanda	0
Descripció	Cosmètica per a les mans

The "Actiu" radio button group shows "Sí" (radio button checked) and "No" (radio button unselected). The "Crea" button at the bottom is also present.

En premer *Crear* us mostrerà la llista de medicaments amb el nou medicament afegit (vegeu la figura 4.3).

FIGURA 4.3. Sortida de la llista amb el medicament afegit

Medicaments			
Llista de medicaments en magatzem			
Ibuprofèn Ibuprofèn 600 mg 2,0 € 214 unitats en magatzem	Paracetamol Paracetamol 1 g 2,6 € 56 unitats en magatzem	Àcid acetilsalicílic Àcid acetilsalicílic 2,6 € 15 unitats en magatzem	Crema Cosmètica per a les mans 12,0 € 12 unitats al magatzem
 Detall	 Detall	 Detall	 Detall

4.2 Estoc de medicaments, llista blanca al formulari

El formulari per afegir medicaments de l'aplicació Estoc de medicaments conté totes les propietats de Medicament. S'hi inclouen tots els camps, i alguns no es poden omplir perquè és impossible tenir la informació en crear el medicament, com per exemple les unitats en comanda (encara no hem pogut fer cap comanda).

Spring MVC permet registrar una llista de propietats (*Whitelist*) que poden obviar-se en una petició d'un formulari.

Si una propietat de la *Whitelist* és a la petició, podrem identificar-la i decidir què fer. Per exemple, llançar una excepció.

Això s'aconsegueix afegint al controlador específic l'anotació `@InitBinder` i la implementació del mètode que es dispararà. En aquest mètode es posarà la llista de propietats sota la restricció esmentada, és a dir, la *Whitelist*.

Modifiqueu `MedicamentController` afegint el mètode per identificar la *WhiteList* (en el vostre cas només és `stockInOrder`). El codi és el que es mostra a continuació:

```

1  @InitBinder
2      public void initialiseBinder(WebDataBinder binder) {
3          binder.setDisallowedFields("stockInOrder");
4      }

```

El codi del projecte "stmedioc" en l'estat **llista blanca al formulari** es pot descarregar des de l'enllaç que trobareu als annexos de la unitat. Però per seguir el desenvolupament és millor partir del que ja heu fet servir i copiar-lo amb el nom de "stmedioc402".

A la mateixa classe, canvieu el mètode `processAddNewMedicamentForm` pel codi següent.

```

1  public String processAddNewMedicamentForm(@ModelAttribute("newMedicament")
2      Medicament newMedicamentToAdd, BindingResult result) {
3          String[] suppressedFields = result.getSuppressedFields();
4          if (suppressedFields.length > 0) {
5              throw new RuntimeException("Attempting to bind disallowed fields: "
6                  + StringUtils.arrayToCommaDelimitedString(suppressedFields));
7          }
8          medicamentService.addMedicament(newMedicamentToAdd);
9          return "redirect:/medicaments/all";
10     }

```

Fixeu-vos que hem afegit el paràmetre `BindingResult result`. Aquest paràmetre ens serveix per determinar els camps continguts a la petició i són a la *Whitelist* (`result.getSuppressedFields`). Recordeu que no voleu que hi siguin, i per això llanceu una excepció.

Si executeu l'aplicació per mostrar el formulari, l'ompliu i l'envieu, haureu d'obtenir l'excepció (vegeu la figura 4.4).

FIGURA 4.4. Sortida per a l'excepció de la 'Whitelist'

```
HTTP Status 500 - Internal Server Error
-----
type Exception report
message Internal Server Error
description The server encountered an internal error that prevented it from fulfilling this request.
exception
org.springframework.web.util.NestedServletException: Request processing failed; nested exception is java.lang.RuntimeException: Intentat accedir amb camps no permesos: stockInOrder
root cause
java.lang.RuntimeException: Intentat accedir amb camps no permesos: stockInOrder
note The full stack traces of the exception and its root causes are available in the GlassFish Server Open Source Edition 4.1 logs.
-----
GlassFish Server Open Source Edition 4.1
```

Traieu aquest camp del formulari de la vista `addMedicament.jsp` i veureu que treballa normalment sense donar l'excepció.

4.3 Estoc de medicaments, pàgina de 'login'

L'aplicació Estoc de medicaments mostra la llista de medicaments i permet crear-ne, però normalment no és convenient que qualsevol usuari connectat pugui afegir-hi medicaments.

Fareu que l'aplicació Estoc de medicaments tingui una pàgina de *login* que es cridi quan l'usuari intenti anar a la pàgina d'afegir medicaments. Aquesta pàgina de *login* autenticarà l'usuari mitjançant un codi d'usuari i una contrasenya, i segons qui sigui l'autoritzarà a anar a la pàgina d'afegir medicaments (**control d'accés**).

Per a aquesta implementació fareu servir funcionalitat Spring Security, afegint l'autenticació bàsica en les nostres pàgines web.

A Estoc de medicaments es fa servir part de la funcionalitat de Spring Security, tot un *framework* que proporciona autenticació i control d'accés configurable:
bit.ly/2nywYZy.

4.3.1 Configurant Spring Security

Spring Security és un *framework* que permet configurar i personalitzar l'autenticació i el control d'accés en aplicacions Spring.

Per fer servir Spring Security a la vostra aplicació heu de configurar les dependències del *framework* que afegiran les llibreries necessàries.

El codi del projecte "stmedioc" en l'estat **pàgina de login** es pot descarregar des de l'enllaç que trobareu als annexos de la unitat. Però per seguir el desenvolupament és millor partir del que ja heu fet servir i copiar-lo amb el nom de "stmedioc403".

Afegiu les dependències que es mostren en el fitxer pom.xml.

```
1 <dependency>
2   <groupId>org.springframework.security</groupId>
```

```

3      <artifactId>spring-security-config</artifactId>
4      <version>3.1.4.RELEASE</version>
5      <exclusions>
6          <exclusion>
7              <artifactId>spring-asn</artifactId>
8              <groupId>org.springframework</groupId>
9          </exclusion>
10         </exclusions>
11     </dependency>
12     <dependency>
13         <groupId>org.springframework.security</groupId>
14         <artifactId>spring-security-web</artifactId>
15         <version>3.1.4.RELEASE</version>
16     </dependency>

```

Recordeu que també les podeu afegir amb l'assistent de NetBeans a tal efecte, la qual cosa us pot ajudar a trobar la versió més recent.

A la carpeta *WEB-INF/spring*, creeu l'arxiu security-context.xml amb el codi que es mostra.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:context="http://www.springframework.org/schema/context"
5      xmlns:mvc="http://www.springframework.org/schema/mvc"
6      xmlns:security="http://www.springframework.org/schema/security"
7      xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.
8          springframework.org/schema/mvc-spring-mvc.xsd
9          http://www.springframework.org/schema/security http://www.
10         springframework.org/schema/security/spring-security.xsd
11         http://www.springframework.org/schema/beans http://www.
12         springframework.org/schema/beans/spring-beans.xsd
13         http://www.springframework.org/schema/context http://www.
14         springframework.org/schema/context/spring-context.xsd">
15     <security:http auto-config="true" use-expressions="true">
16         <security:intercept-url pattern="/medicaments/add" access="hasRole('
17             adminRol')"/>
18         <security:form-login login-page="/login" default-target-url="/
19             medicaments/add" login-processing-url="/j_spring_security_check"
20                 username-parameter="j_username" password-parameter
21                 ="j_password" authentication-failure-url="/
22                 loginfailed"/>
23         <security:logout logout-success-url="/logout" />
24     </security:http>
25     <security:authentication-manager>
26         <security:authentication-provider>
27             <security:user-service>
28                 <security:user name="ioc" password="ioc123" authorities="
29                     adminRol" />
30             </security:user-service>
31         </security:authentication-provider>
32     </security:authentication-manager>
33 </beans>

```

L'etiqueta **security:http**, els seus atributs i el seu contingut defineixen el veritable comportament de la seguretat.

L'etiqueta i els atributs

```

1  <security:intercept-url pattern="/medicaments/add" access="hasRole('adminRol')
2      "/>

```

indiquen que una petició sobre */medicaments/add* serà interceptada i només hi podrà accedir qui hagi estat identificat amb *rol adminRol*.

L'etiqueta i els atributs

```
1 <security:form-login ...>
```

defineixen les pàgines que controlen l'autenticació efectiva, la fallida i el nom dels camps de *login*.

L'etiqueta i l'atribut

```
1 <security:logout logout-success-url="/logout" />
```

defineixen la pàgina de desconexió.

L'etiqueta i els atributs

```
1 <security:user name="ioc" password="ioc123" authorities="adminRol" />
```

defineixen els usuaris i les seves credencials. L'atribut *authorities* determina el rol de l'usuari. En aquest cas només heu configurat un usuari, però se'n poden configurar més repetint l'etiqueta.

Completeu la configuració a nivell de l'aplicació. A web.xml, afegiu la següent configuració dins de <web-app>:

```
1 <context-param>
2     <param-name>contextConfigLocation</param-name>
3     <param-value>/WEB-INF/spring/security-context.xml</param-value>
4 </context-param>
5 <listener>
6     <listener-class>org.springframework.web.context.ContextLoaderListener</
7         listener-class>
8 </listener>
9 <filter>
10    <filter-name>springSecurityFilterChain</filter-name>
11    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</
12        filter-class>
13 </filter>
14 <filter-mapping>
15     <filter-name>springSecurityFilterChain</filter-name>
16     <url-pattern>/*</url-pattern>
17 </filter-mapping>
```

Amb el paràmetre de context <context-param> esteu indicant el nom del fitxer on heu configurat la seguretat. Com que el *listener* és ContextLoaderListener, es carregarà la configuració indicada a security-context.xml a l'inici de l'aplicació i en aquest nivell, sense esperar cap petició i sense dependre del Dispatcher Servlet.

Sense entrar en més detalls de Spring Security, el filtre està indicant que qualsevol petició (<url-pattern>) serà manegada pel filtre springSecurityFilterChain, i fent servir DelegatingFilterProxy esteu deixant a Spring que crei els beans necessaris.

4.3.2 El controlador per al 'login'

A la configuració que heu fet a security-context.xml heu assignat les URL que corresponen al *login*, al *logout* i a les credencials errònies. Són, respectivament, */login*, */logout* i */loginfailed*.

Amb aquesta informació, al paquet cat.xtec.ioc.controller podeu crear un nou controlador LoginController amb el codi que es mostra a continuació:

```

1 package cat.xtec.ioc.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7
8 @Controller
9 public class LoginController {
10
11     @RequestMapping(value = "/login", method = RequestMethod.GET)
12     public String login() {
13         return "login";
14     }
15
16     @RequestMapping(value = "/loginfailed", method = RequestMethod.GET)
17     public String loginerror(Model model) {
18         model.addAttribute("error", "true");
19         return "login";
20     }
21
22     @RequestMapping(value = "/logout", method = RequestMethod.GET)
23     public String logout(Model model) {
24         return "login";
25     }
26 }
```

Quan algú intenti accedir a l'URL */medicaments/add*, Spring Security mostrarà la pàgina login.jsp. Si l'usuari no s'autentifica o no està en el rol adient, segons la configuració actuarà */loginfailed*, i s'hi afegirà l'atribut **error**.

Si l'usuari provoca una desconnexió, llavors actuarà */logout* i es mostrarà un altre cop la pàgina de *login*.

4.3.3 La vistes

Com que un cop connectats l'aplicació us dirigirà a la vista addMedicament, podeu afegir aquí la desconnexió.

Afegiu el codi següent a addMedicament.jsp dins de la *div* de classe jumbotron.

```

1 <a href="

```

A *views*, creeu la vista login.jsp amb el codi que es mostra a continuació:

```

1 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
2 <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
3 <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
4 <%@page contentType="text/html" pageEncoding="UTF-8"%>
5 <!DOCTYPE html>
6 <html>
7   <head>
8     <meta http-equiv="Content-Type" content="text/html; charset=ISO
9       -8859-1">
10    <link
11      rel="stylesheet" href="//netdna.bootstrapcdncdn.com/bootstrap/3.0.0/css
12        /bootstrap.min.css">
13    <title>Medicaments</title>
14  </head>
15  <body>
16    <section>
17      <div class="jumbotron">
18        <div class="container">
19          <h1>Medicament</h1>
20          <p>Afegir medicament</p>
21        </div>
22      </div>
23    </section>
24    <div class="container">
25      <div class="row">
26        <div class="col-md-4 col-md-offset-4">
27          <div class="panel panel-default">
28            <div class="panel-heading">
29              <h3 class="panel-title">Si us plau, proporcioni les
30                seves dades</h3>
31            </div>
32            <div class="panel-body">
33              <c:if test="${not empty error}">
34                <div class="alert alert-danger">
35                  Credencials incorrectes
36                </div>
37              </c:if>
38              <form action=<c:url value=
39                j_spring_security_check> </c:url>" method="post">
40                <fieldset>
41                  <div class="form-group">
42                    <input class="form-control" placeholder
43                      ="Usuari" name='j_username' type="text">
44                  </div>
45                  <div class="form-group">
46                    <input class="form-control" placeholder
47                      ="Contrasenya" name='j_password'
48                      type="password">
49                  </div>
50                  <input class="btn btn-lg btn-success btn-block"
51                      type="submit" value="Connectar
52                      ">
53                </fieldset>
54              </form>
55            </div>
56          </div>
57        </div>
58      </div>
59    </body>

```

L'estructura de selecció `<c:if ...>` mostrarà el missatge en el cas que l'atribut `error` no sigui buit. Recordeu que el controlador l'omplirà en cas que l'usuari no s'hagi autenticat.

L'acció del formulari es correspon a la configuració feta a security-context.xml, concretament l'atribut `login-processing-url=''/j_spring_security_check'`, i els noms dels camps “usuari” i “contrasenya” també es corresponen amb la configuració.

4.3.4 Provant d'entrar

Executeu l'aplicació amb l'URL `localhost:8080/stmedioc403/medicaments/add`.

Veureu la pàgina de *login* tal com es mostra en la figura 4.5.

FIGURA 4.5. Formulari de 'login' (autenticació)

El formulari de login està dividit en dues parts: un encapçalament blau superior amb el text "Medicament" i "Afegeix medicament" i un contingut gris inferior. El contingut gris conté el missatge "Si us plau, proporcioneu les vostres dades" i dos camp d'entrada per "Usuari" i "Contrassenya". A baix, hi ha un botó verd "Connecta".

Proveu amb dades no correctes i l'aplicació us mostrarà el missatge de credencials no vàlides (vegeu la figura 4.6).

FIGURA 4.6. Formulari de 'login' (autenticació), credencials incorrectes

El formulari de login està dividit en dues parts: un encapçalament blau superior amb el text "Medicament" i "Afegeix medicament" i un contingut gris inferior. El contingut gris conté el missatge "Si us plau, proporcioneu les vostres dades" i un missatge d'error "Credencials incorrectes" en un rectangle vermell. Abans d'aquest missatge, hi havia els camps d'entrada "Usuari" i "Contrassenya". A baix, hi ha un botó verd "Connecta".

Proveu, doncs, amb credencials correctes, i llavors us mostrarà la pàgina per afegir el medicament. Fixeu-vos que hi ha l'enllaç a desconnexió (vegeu la figura 4.7).

FIGURA 4.7. Formulari de medicament amb enllaç a desconexió

The screenshot shows a web-based form for adding a new medicine. The title bar says 'Medicament'. Below it, there's a button labeled 'Afegeix medicament' and another labeled 'desconnecta'. The main area contains fields for 'Codí', 'Nom', 'Preu' (with a value of '0.0'), 'Laboratori', 'Categoria', 'Unitats en estoc' (with a value of '0'), and 'Descripció'. Below these fields are two radio buttons for 'Actiu': 'Sí' (unchecked) and 'No' (checked). At the bottom is a blue button labeled 'Crea'.

Amb aquest exemple heu assegurat una pàgina de la vostra aplicació, i es pot interpretar la generalització a qualsevol pàgina de l'aplicació.

Heu fet servir **Spring Security** baixant les dependències i l'heu configurat amb l'arxiu security-context.xml, on heu definit el comportament de Spring Security quant a:

- Quines pàgines s'ha d'assegurar (interceptar) i quins rols estaran autoritzats a accedir-hi.
- Quines són les pàgines i URL de *login*, *logout* i *error*.
- Quin és el formulari que feu servir per a l'autenticació.
- Quins són els camps d'usuari i contrasenya que conté el formulari.
- Quins usuaris hi ha a l'aplicació.

La configuració d'aquest arxiu i el *bean* que manega la seguretat es carreguen a l'inici de l'aplicació, i serà vàlida per a tot el context de l'aplicació segons el que s'ha indicat a web.xml.

Heu implementat un controlador que manega les peticions segons les URL de l'arxiu de configuració.

Finalment, heu implementat la vista amb el formulari d'autenticació (*login*).

4.4 Estoc de medicaments, internacionalització

L'aplicació Estoc de medicaments està practicament completada segons els objectius pedagògics que s'havien proposat. Però avui dia les aplicacions han de ser

multiidioma, i per això cal dotar la nostra aplicació de la possibilitat de mostrar-se en diversos idiomes.

No tan sols això, sinó que es parlarà d'una **internacionalització** que permet definir més paràmetres, com quina moneda es fa servir i quin és el separador decimal, entre altres personalitzacions.

Fareu servir la funcionalitat que es coneix com a **externalització de missatges** a Spring MVC per mostrar el text de les etiquetes en diversos idiomes, segons arxius de propietats.

L'externalització és el pas previ a la internacionalització, ja que cal tenir el text de les etiquetes dels diferents idiomes en diferents arxius de configuració per tal que la funcionalitat d'*i18N* vagi a buscar-los.

Un cop externalitzats els missatges, ja es pot configurar i implementar tot el que és relatiu a *i18n*. Per detectar i canviar d'idioma fareu servir SessionLocaleResolver a nivell de *web application context* i l'interceptor LocaleChangeInterceptor també en aquest nivell.

La **internacionalització** s'abreia sovint com "i18n" (de l'anglès I-eighteen letters-N), que prové de la "i" del començament, les 18 lletres següents i la "n" del final.

El codi del projecte "stmedioc" en l'estat **internacionalització** es pot descarregar des de l'enllaç que trobareu als annexos de la unitat.
Però per seguir el desenvolupament és millor partir del que ja heu fet servir i copiar-lo amb el nom de "stmedioc404".

4.4.1 Externalitzem els missatges

Per externalitzar els missatges de la vostra aplicació i després poder traduir-los i fer-los servir en diversos idiomes fareu canvis en diversos textos d'algunes etiquetes del formulari de la vista addMedicament.jsp.

Per altra banda, creareu diversos fitxers de propietats, un per defecte i altres per a altres idiomes (concretament, un addicional per a l'anglès).

A més, haureu de configurar un *bean* de Spring per connectar la vista amb els fitxers de propietats que contenen els textos.

A addMedicament.jsp, modifiqueu les etiquetes de *code*, *name*, *price*, *producer* i *category* substituint el text, respectivament, per les etiquetes Spring que s'hi mostren.

```

1 <spring:message code= "addMedicament.form.medicamentId.label"/>
2 <spring:message code= "addMedicament.form.name.label"/>
3 <spring:message code= "addMedicament.form.price.label"/>
4 <spring:message code= "addMedicament.form.producer.label"/>
5 <spring:message code= "addMedicament.form.category.label"/>
```

Amb l'etiqueta *spring:message*, mitjançant l'atribut *code*, esteu associant el text amb la propietat definida al fitxer de propietats que creareu per a cada idioma de l'aplicació.

messages.properties

A la carpeta `/src/main/resources` (el que veieu dins d'*Other Sources* en la pestanya de projectes de NetBeans), creeu un arxiu de propietats de nom `messages.properties`. Serà l'arxiu per defecte, és a dir, que quan no trobi cap idioma serà el que farà servir.

Afegiu-hi les propietats de cada etiqueta.

```

1 addMedicament.form.medicamentId.label = Codi de medicament
2 addMedicament.form.name.label = Nom
3 addMedicament.form.price.label = Preu
4 addMedicament.form.producer.label = Laboratori - Fabricant
5 addMedicament.form.category.label = Categoria

```

Finalment, heu de configurar el *bean* que connecta el fitxer de propietats amb les etiquetes `spring:message` que heu afegit al jsp. Es configura a nivell de *web application context*, és a dir, afegiu un *bean* a l'arxiu `DispatcherServlet-servlet.xml`.

```

1 <bean id= "messageSource" class="org.springframework.context.support.
   ResourceBundleMessageSource">
2   <property name="basename" value="messages"/>
3 </bean>

```

L'atribut `value="messages"` és el que determina que el fitxer de propietats és `messages`.

Executeu l'aplicació amb l'URL `localhost:8080/stmedioc404/medicaments/add` per mostrar el formulari. Això sí, us haureu d'identificar i, un cop fet, us mostrarà el formulari amb els valors de l'arxiu de propietats (vegeu la figura 4.8).

FIGURA 4.8. Formulari amb textos externalitzats

4.4.2 Configurant i implementant el canvi d'idioma

A *resources*, creeu l'arxiu *messages_en.properties* per poder tenir dos idiomes. Afegiu-hi les propietats de *messages.properties*, però canvieu els valors.

```

1 addMedicament.form.medicamentId.label = Medicament ID
2 addMedicament.form.name.label = Name
3 addMedicament.form.price.label = Price
4 addMedicament.form.producer.label = Producer
5 addMedicament.form.category.label = Category

```

A la vista *addMedicament.jsp*, afegiu el **selector** per canviar d'idioma just després de l'enllaç de desconexió.

```

1 <div class="pull-right" style="padding-right:50px">
2   <a href="?language=ca" >Català</a>|<a href="?language=en" >Anglès</a>
3 </div>

```

A l'arxiu de configuració de *web application context* *DispatcherServlet-servlet.xml*, afegiu la definició del *bean*.

```

1 <bean id="localeResolver" class="org.springframework.web.servlet.i18n.
  SessionLocaleResolver">
2   <property name="defaultLocale" value="ca"/>
3 </bean>

```

SessionLocaleResolver és l'objecte que assigna atributs locals sobre la sessió de l'usuari. Una de les propietats que conté l'objecte és *defaultLocale*. En el vostre cas, dieu que el llenguatge per defecte és el català (*ca*).

També a *DispatcherServlet-servlet.xml*, heu d'afegir l'interceptor *LocaleChangeInterceptor* tal com es mostra a continuació:

```

1 <mvc:interceptors>
2   <bean class="org.springframework.web.servlet.i18n.
    LocaleChangeInterceptor">
3     <property name="paramName" value="language"/>
4   </bean>
5 </mvc:interceptors>

```

Amb aquest interceptor esteu dient que el nom del paràmetre que correspon al canvi d'idioma és *language*, i això es correspon amb el que heu afegit a *addMedicament.jsp*.

4.4.3 Provant a canviar d'idioma

Executeu l'aplicació amb l'URL *localhost:8080/stmedioc404/medicaments/add*, i, un cop autenticats satisfactoriament, us mostrarà el formulari (vegeu la figura 4.9).

FIGURA 4.9. Formulari amb els textos en l'idioma per defecte

Medicament

Afegeix medicament

Català | Anglès disconnecta

Afegeix medicament

Codi

Nom

Preu 0.0

Laboratori

Categoria

Unitats en estoc 0

Descripció

Actiu Sí No

Crea

Fixeu-vos que les etiquetes són en català. Si premeu l'enllaç *Anglès*, llavors les etiquetes canviaran (vegeu la figura 4.10).

FIGURA 4.10. Formulari amb els textos en anglès

Medicament

Afegeix medicament

Català | Anglès disconnecta

Afegeix medicament

MedicamentID

Name

Price 0.0

Producer

Category

Unitats en estoc 0

Descripció

Actiu Sí No

Crea

Si torneu a premer *Català* tornarà a mostrar les etiquetes en aquest idioma. També podeu veure que les URL contenen el paràmetre *language*.

4.5 Què s'ha après?

Per veure com es genera i manega a Spring MVC un flux de dades des de la vista fins al model heu afegit el formulari per crear medicaments.

Al controlador, al mètode que desencadena mostrar el formulari, es crea un atribut que passarà a la resposta amb un *domain object* (*Medicament*) buit. A la vista, a l'etiqueta `form:form`, mitjançant l'atribut d'etiqueta `modelAttribute`,

es relacionen els camps del formulari amb les propietats de l'objecte. Canvis en els valors del formulari provoquen canvis en l'objecte, i per això s'anomena *form-backing bean*.

El mètode que recull el POST del formulari amb l'anotació:

```
1 @ModelAttribute("newMedicament") Medicament newMedicamentToAdd
```

està recollint l'objecte *form-backing bean* i el podrà fer servir amb la variable declarada (`newMedicamentToAdd`).

Heu afegit una llista de propietats (*Whitelist*) per obviar un dels camps del model (`stockInOrder`). Ho heu aconseguit amb l'anotació `@InitBinder` i la implementació del mètode que la segueix al controlador específic.

Heu fet servir **Spring Security**, un *framework* que permet configurar i personalitzar l'autenticació i el control d'accés en aplicacions Spring. Per usar el *framework* heu desenvolupat la pàgina de *login*, afegint l'arxiu de configuració de la seguretat `security-context.xml`, on heu definit els usuaris i el comportament de la vostra aplicació respecte a la seguretat (per exemple, quines pàgines necessitaran autenticació i quines pàgines han de mostrar-se per a *login*, *logout* i *error*).

Finalment, heu fet que la vostra aplicació tingui **internacionalització** (*i18n*). Heu externalitzat missatges ajudats pels fitxers de propietats i etiquetes específiques als *jsp* (`spring:message`) i heu configurat la internacionalització amb els *beans* necessaris a nivell de *web application context*: `SessionLocaleResolver` i `LocaleChangeInterceptor`.

Tècniques d'accés a dades

Raúl Velaz Mayo

Desenvolupament web en entorn servidor

Índex

Introducció	5
Resultats d'aprenentatge	7
1 Accés a dades amb JDBC	9
1.1 Importar un projecte de Maven a Netbeans	9
1.2 La primera connexió a una base de dades	10
1.3 Millorant el codi: fitxers de propietats i tests unitaris	12
1.4 Tests unitaris amb JUnit	16
1.4.1 El 'framework' de test JUnit	17
1.4.2 Cobertura dels tests unitaris	18
1.4.3 Augmentant la cobertura del codi	20
1.5 Fent consultes a la base de dades	22
1.6 Operacions CRUD	28
1.6.1 Operacions de lectura	29
1.6.2 Operacions d'escriptura	34
1.6.3 Eliminació de dades	39
1.7 Injecció SQL	39
1.8 Què s'ha après?	42
2 Accés a dades amb Java Enterprise Edition	43
2.1 "SocIoc". Dialogant amb clients amb JPA	44
2.1.1 Anotacions JPA	49
2.1.2 Unitats de persistència	53
2.2 Servidor d'aplicacions Glassfish	58
2.2.1 'Pool' de connexions	59
2.2.2 Recursos JDBC	62
2.2.3 Connectar l'aplicació "SocIoc" amb el recurs JDBC	63
2.2.4 Refactoritzant el codi per simplificar el codi i millorar el test	70
2.3 Què s'ha après?	74
3 Accés a dades amb Spring i Hibernate	77
3.1 "SocIoc". Dialogant amb usuaris amb Spring i Hibernate	78
3.1.1 Integrant l'aplicació "SocIoc" amb Spring	80
3.1.2 Integrant l'aplicació "SocIoc" amb Hibernate	85
3.2 "SocIoc". Dialogant amb preguntes i respostes	101
3.3 "SocIoc". Dialogant amb usuaris, rangs i vots	113
3.4 Què s'ha après?	121

Introducció

Tot i que la finalitat i la diversitat d'aplicacions que existeixen és difícilment classificable, sí que es pot dir que totes tenen en comú uns components bàsics. Una lògica de negoci que determinarà la funció del programari, les connexions amb sistemes externs, una sèrie d'interfícies d'usuari i les dades que han de ser guardades en bases de dades amb la capacitat de ser modificades. La diversitat d'aplicacions està relacionada amb la diversitat de bases de dades (BD) que existeixen, i en funció de l'aplicació que esteu construint haureu d'escol·lir una base de dades o una altra. Això podria presentar un problema si per a cada tipus de base de dades haguéssiu d'escriure un codi diferent. Imagineu, per exemple, que dissenyeu una aplicació que utilitza MySQL com a base de dades però que uns mesos després canviem els requeriments i decidim emprar Postgresql. Idealment voldríeu fer la transició a la nova base de dades amb els mínims canvis possibles, és a dir, el que voleu és que la base de dades estigui desacoblada de l'aplicació. Per aconseguir aquesta independència existeix JDBC (Java DataBase Connectivity), que és una API (Application Programming Interface) que defineix com els clients (aplicacions Java) accedeixen a les bases de dades. Proporciona una interfície de programació comú que permet que les aplicacions Java accedeixin a les bases de dades sense preocupar-se dels detalls particulars de cada BD.

Explicarem com connectar les nostres aplicacions a bases de dades mitjançant JDBC i farem tests per comprovar la connexió i les diferents consultes i actualitzacions de les dades.

Continuarem amb la utilització de Java Persistence API (JPA), una interfície de Java que ens permetrà tenir un lleguatge de consulta independent de la base de dades. Primer farem servir la implementació de JPA EclipseLink directament amb Java Enterprise Edition, configurant la connexió sobre el servidor d'aplicacions Glassfish.

Finalment, farem servir la implementació de JPA Hibernate amb el *framework* Spring. Per una banda, Hibernate ens facilitarà la connexió a la base de dades, i Spring ens ajudarà en el disseny de l'aplicació.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Desenvolupa aplicacions d'accés a magatzems de dades, aplicant mesures per mantenir la seguretat i la integritat de la informació.

- Analitza les tecnologies que permeten l'accés mitjançant programació a la informació disponible en magatzems de dades.
- Crea aplicacions que estableixin connexions amb bases de dades.
- Recupera informació emmagatzemada en bases de dades.
- Publica en aplicacions web la informació recuperada.
- Utilitza conjunts de dades per emmagatzemar la informació.
- Crea aplicacions web que permetin l'actualització i l'eliminació d'informació disponible en una base de dades.
- Utilitza transaccions per mantenir la consistència de la informació.
- Prova i documenta les aplicacions.

1. Accés a dades amb JDBC

Comencem a desenvolupar un projecte que anomenarem “SocIoc” (SOCial IOC) i que consisteix en una xarxa social de preguntes i respostes per a estudiants de l’IOC. La idea està basada en stackoverflow.com, una xarxa social on els desenvolupadors de programari poden fer preguntes de desenvolupament que són contestades per altres desenvolupadors. La funcionalitat és simple: els estudiants de l’IOC poden fer i contestar preguntes i votar positiva o negativament les respostes a cada pregunta. Els estudiants que rebin vots positius a les respostes aniran guanyant punts.

En aquesta unitat posem les bases de l’aprenentatge configurant una aplicació Java per establir connexions amb la BD, executar consultes SQL (Structured Query Language) i tancar la connexió. Explicarem:

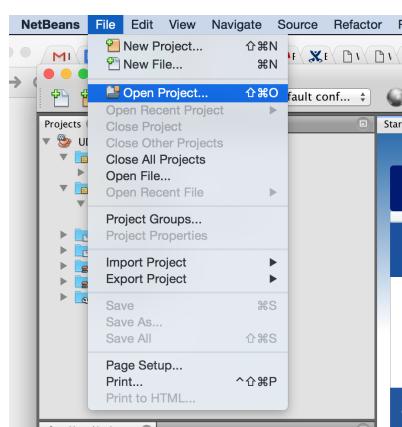
- Com fer connexions a la base de dades H2 utilitzant *drivers* JDBC.
- Com escriure tests unitaris per comprovar que el codi funciona correctament.
- Com fer operacions de lectura, escriptura, actualització i esborrat de base de dades.
- Com preveure injeccions malicioses de codi SQL quan fem consultes a les bases de dades.

1.1 Importar un projecte de Maven a Netbeans

En la figura 1.1 es mostra on és l’opció per obrir un projecte existent.

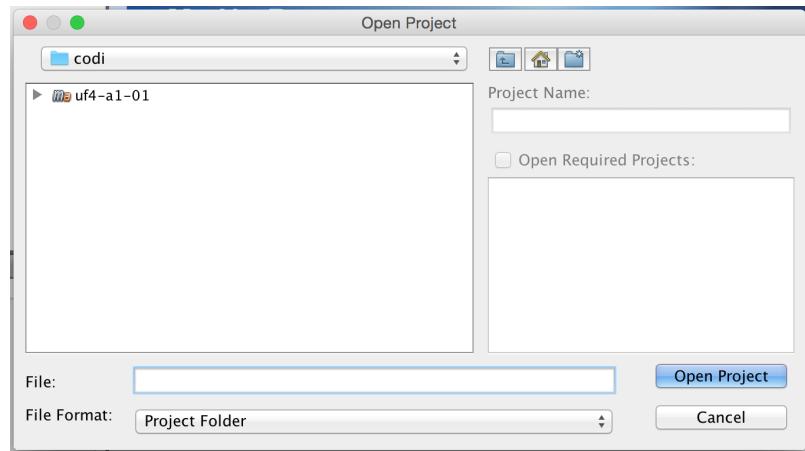
El projecte amb el qual treballareu el podeu trobar a l’apartat d’annexos de la unitat. Un cop descarregat el fitxer i descomprimit ja el podeu importar.

FIGURA 1.1. Obrir un projecte



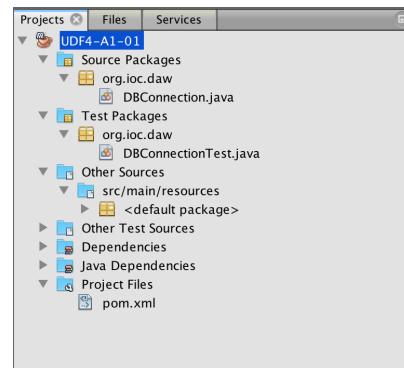
Navegueu per les carpetes fins on teniu la carpeta amb el projecte. Com podeu veure en la figura 1.2, Netbeans reconeixerà que és un projecte Maven.

FIGURA 1.2. Seleccionar el projecte



Finalment, un cop obert el projecte, Maven començarà a descarregar les dependències de JUnit i el *driver* d'H2. Hauríeu de veure un projecte amb l'estruccura que es mostra en la figura 1.3.

FIGURA 1.3. Estructura del projecte



1.2 La primera connexió a una base de dades

Apache Maven permet obtenir les llibreries necessàries per al projecte que veureu a continuació, on simplement fareu una connexió a la BD en memòria H2.

En el fitxer pom.xml de Maven, les úniques dependències que us fan falta són la del *driver* de la BD H2 i de JUnit, que és un conjunt de llibreries que utilitzareu per escriure els tests unitaris.

```

1 <dependencies>
2   <dependency>
3     <groupId>com.h2database</groupId>
4     <artifactId>h2</artifactId>
5     <version>1.4.190</version>
6   </dependency>
7   <!-- testing -->
8   <dependency>
```

```

9      <groupId>junit</groupId>
10     <artifactId>junit</artifactId>
11     <version>4.12</version>
12   </dependency>
13 </dependencies>
```

Un cop Maven descarrega les dependències i les afegeix al *classpath*, ja podeu escriure la primera classe, que simplement establirà una connexió amb una base de dades.

```

1 public Connection getConnection() {
2     Connection con = null;
3     try {
4         Class.forName("org.h2.Driver");
5         con = DriverManager.getConnection("jdbc:h2:mem:socioc_db", "usuari" , "passwd");
6     } catch (ClassNotFoundException | SQLException e) {
7         e.printStackTrace();
8     }
9     return con;
10 }
```

Cada *driver* JDBC té una classe que s'encarrega d'inicialitzar el *driver* quan es carrega a memòria. En el cas d'una BD H2:

```
1 Class.forName("org.h2.Driver");
```

Carregar el *driver* en memòria no és necessari, ja que des de la versió 6 de Java es fa automàticament. Això implica que com que estem utilitzant una versió de Java superior a la 6, no fa falta que carreguem el *driver* explícitament. Quan, més tard, refactoritzarem el codi, podrem eliminar *Class.forName("org.h2.Driver")*.

Un cop el *driver* està carregat en la memòria es pot procedir a connectar amb la BD. Per obrir una connexió amb la BD s'utilitza la classe *java.sql.DriverManager*.

```
1 con = DriverManager.getConnection("jdbc:h2:mem:socioc_db");
```

Fixeu-vos en el paràmetre que s'ha utilitzat per establir la connexió:

```
1 jdbc:h2:mem:socioc_db
```

Intenteu contestar a les següents preguntes:

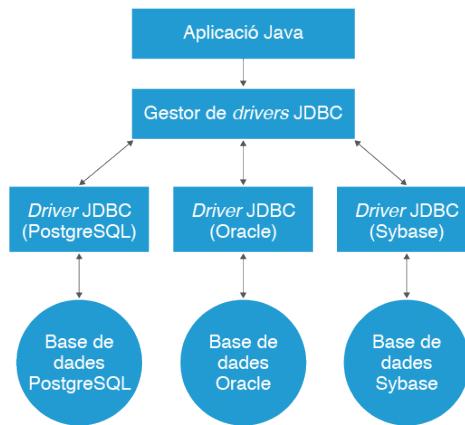
- Quina és l'estructura d'aquest paràmetre de connexió?
- Què representa el paràmetre *socioc_db*?

La resposta està relacionada amb el disseny de JDBC i amb la seva arquitectura, que té tres components principals que podeu veure en la figura 1.4:

- Gestor de *drivers* (*DriverManager*): és el responsable de trobar el *driver* que l'aplicació necessita. Quan es sol·licita una connexió amb la BD es fa mitjançant un URL (Uniform Resource Locator), que descriu com ha de ser la connexió.

Algunes de les bases de dades compatibles amb JDBC són: DB2, H2, Informix, JavaDB/Derby, Microsoft SQL Server, Mimer SQL, MySQL, NuoDB, Oracle, PostgreSQL, SQLite, Sybase i Vertica.

- Driver JDBC: cada *driver* present al sistema es carrega en la màquina virtual de Java (JVM Java Virtual Machine) i es registra amb el DriverManager. Quan una aplicació sol·licita una connexió amb una BD, el DriverManager s'encarrega de preguntar a cada *driver* present si pot connectar amb la BD amb l'URL de connexió especificat.
- Base de dades compatibles amb JDBC.

FIGURA 1.4. Arquitectura JDBC

L'URL de connexió amb una BD JDBC té el següent format:

¹ `jdbc:Tipus_de_base_de_dades://<Host>:<Port>/<Base_de_dades>`

Per tant, la URL de connexió `jdbc:h2:mem:socioc_db` indica el següent:

- jdbc: indica que s'utilitzarà JDBC per fer la connexió.
- h2: és el tipus de servidor de BD amb el qual volem connectar.
- mem: en aquest cas, com que és una BD en memòria no s'ha d'especificar ni la IP ni el port on està corrent la BD.
- socioc_db: és el nom de l'esquema que s'utilitzarà.

1.3 Millorant el codi: fitxers de propietats i tests unitaris

Hi ha dues coses al codi de l'apartat 1.2 que podeu millorar. La primera és que no és necessari carregar el *driver* explícitament. La segona és que mai és una bona idea posar l'URL de connexió amb la BD directament a una classe. La gràcia d'utilitzar JDBC és que l'aplicació serà independent de la BD que utilitzem; si posem el valor de l'URL de connexió estarem trencant això, ja que en canviar de BD haurem de modificar la classe. Això es pot evitar creant un fitxer de propietats on es configuri els paràmetres de connexió amb la BD. El format del fitxer és PROPIETAT=VALOR, com podeu veure:

```

1 DB_DRIVER_CLASS=org.h2.Driver
2 DB_URL=jdbc:h2:mem:socio_db
3 DB_USERNAME=usuari
4 DB_PASSWORD=passwd

```

El fitxer de propietats s'anomena db.properties i es troba a *src/main/resources*. Un cop el fitxer de propietats està definit ja el podeu utilitzar per llegir les propietats de configuració.

```

1 public Connection getConnection() throws SQLException, IOException {
2     Properties props = new Properties();
3     InputStream resourceAsStream = null;
4     Connection con = null;
5     try {
6         ClassLoader classLoader = getClass().getClassLoader();
7         URL urlResource = classLoader.getResource("db.properties");
8         if(urlResource != null){
9             resourceAsStream = urlResource.openStream();
10            props.load(resourceAsStream);
11            con = DriverManager.getConnection(props.getProperty("DB_URL"),
12                props.getProperty("DB_USERNAME"),
13                props.getProperty("DB_PASSWORD"));
14        }
15    } catch (IOException | ClassNotFoundException | SQLException e) {
16        e.printStackTrace();
17    } finally {
18        if (resourceAsStream != null) {
19            resourceAsStream.close();
20        }
21    }
22    return con;
23 }

```

Si volguéssiu canviar la BD, l'únic que s'hauria de fer és canviar el contingut del fitxer de propietats. D'aquesta manera aconseguireu que el codi estigui feblement acoblat (*loosely coupled*) amb la configuració.

Un cop s'ha llegit el fitxer que conté les propietats amb el codi:

```

1 URL urlResource = classLoader.getResource("db.properties");

```

La classe `java.util.Properties` permet recuperar el valor de les propietats:

```

1 props.getProperty(NOM_DE_LA_PROPRIETAT)

```

Un cop tenim la classe, escriurem el test unitari.

```

1 public class DBConnectionTest {
2     DBConnection dBConnection;
3     Connection connection;
4
5     @Before
6     public void setUp(){
7         dBConnection = new DBConnection();
8     }
9
10    @After
11    public void cleanUp() throws SQLException {
12        connection.close();
13    }
14
15    @Test

```

```

16     public void conectarAmbLaBaseDeDades() throws IOException, SQLException {
17         connection = dBConnection.getConnection();
18         Assert.assertEquals("H2 JDBC Driver", connection.getMetaData().
19             getDriverName());
20         Assert.assertEquals("SOCIOC_DB", connection.getCatalog());
21     }

```

En aquest primer test només esteu comprovant dues coses: que el nom de l'esquema que hem definit a l'URL de connexió és SOCIOC_DB i que el *driver* que esteu utilitzant és el *driver* JDBC d'H2. El mètode assertEquals de JUnit permet comprovar que l'objecte esperat (primer argument) és igual que el que s'obté quan s'executa el nostre codi.

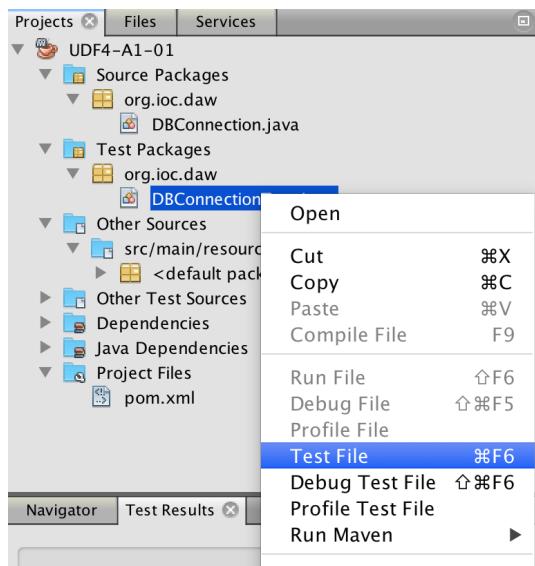
```

1  Assert.assertEquals("H2 JDBC Driver", connection.getMetaData().getDriverName())
2      ;
2  Assert.assertEquals("SOCIOC_DB", connection.getCatalog());

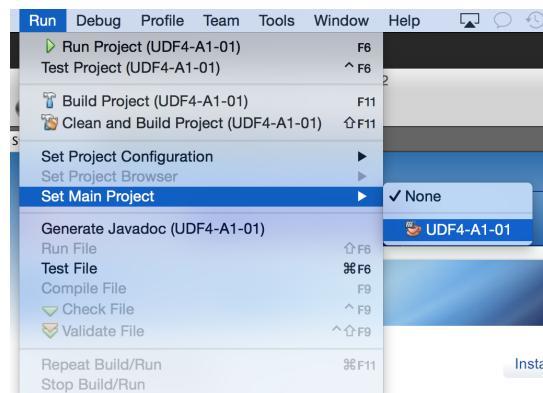
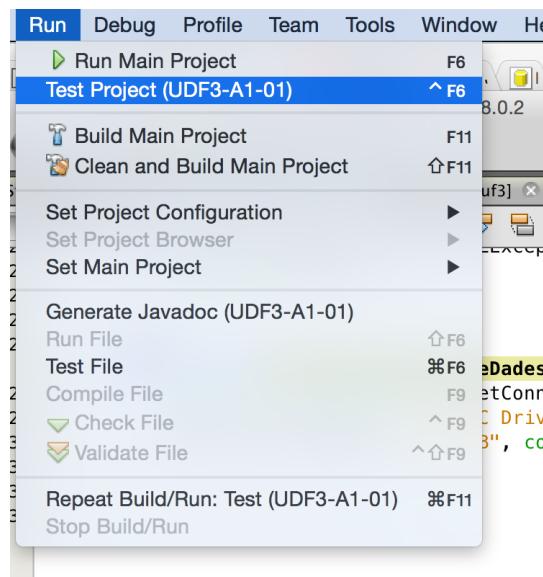
```

A Netbeans, els tests es poden executar de forma individual, és a dir, classe per classe o tots els del projecte. Tal com podeu veure en la figura 1.5, si feu clic amb el botó dret sobre una classe dins de la carpeta *Test Packages* podreu executar els tests per a la classe seleccionada.

FIGURA 1.5. Execució dels tests d'una classe



Per executar tots els tests d'un projecte, primer heu de designar com a projecte principal el projecte UDF4-A1-01, tal com podeu veure en la figura 1.6. A continuació podreu executar tots els tests del projecte, tal com s'indica en la figura 1.7.

FIGURA 1.6. Selecció del projecte principal**FIGURA 1.7.** Selecció del projecte principal

Quan executeu els tests de JUnit a Netbeans, els resultats es mostren a la finestra de resultats de la figura 1.8. Per mostrar aquesta finestra segui els passos mostrats en figura 1.9.

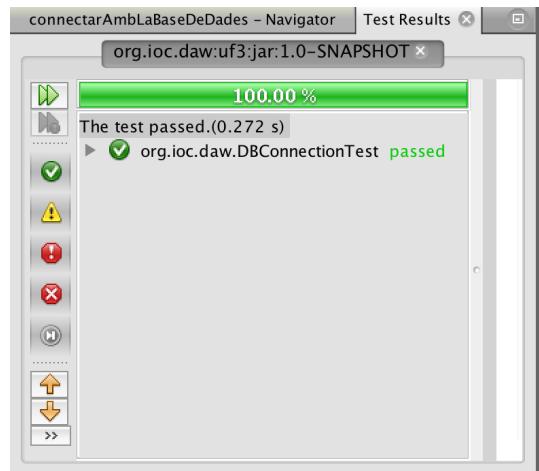
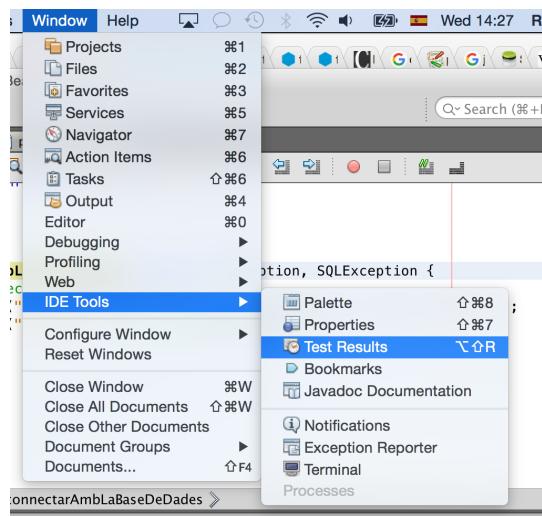
FIGURA 1.8. Finestra de resultat de tests

FIGURA 1.9. Mostrar finestra de resultat de tests

1.4 Tests unitaris amb JUnit

Els tests unitaris o proves de components són un tipus de proves de programari que consisteixen a fer proves sobre els components o unitats més petits del codi font d'una aplicació o d'un sistema. Això dóna la capacitat de verificar que les vostres funcions funcionen com s'esperava. És a dir, que per a qualsevol funció, i donat un conjunt d'entrades, podeu determinar si la funció retorna els valors adients i tracta correctament els errors.

Això ajuda a identificar les falles en els nostres algoritmes i/o lògica i ajuda a millorar la qualitat del codi que comprèn una funció determinada. Absolutament tots els components d'una aplicació han de tenir tests unitaris, fet que permet, durant qualsevol moment durant el desenvolupament, verificar la qualitat del treball.

Un segon avantatge de desenvolupar codi pensant sempre que s'ha de poder testejar és que el codi resultant és més fàcil de testejar. Com a resultat s'acaba estructurant el codi millor i es creen un major nombre de funcions més petites i especialitzades.

Un tercer avantatge de tenir un conjunt de tests unitaris sòlids és que preveuen que futurs canvis al codi trenquin la funcionalitat. Si en fer un canvi i executar els tests hi ha un error és clar que els canvis han introduït un error a la part específica que el test unitari està comprovant. Finalment, els tests unitaris proporcionen la millor documentació del sistema, ja que reflecteix exactament què s'espera que faci el codi. Els desenvolupadors que vulguin aprendre quina funcionalitat proporciona cada un dels components del sistema només han de llegir els tests unitaris.

1.4.1 El 'framework' de test JUnit

JUnit en la versió 4.x és un *framework* de test que utilitza anotacions per identificar els mètodes que especificuen un test. Un test unitari és un mètode que s'especifica en una classe que només s'utilitza per al test. Això s'anomena *classe de test*. Per definir un mètode de test amb el *framework* JUnit 4.x es fa amb l'anotació `@org.junit.Test`. En aquest mètode s'utilitza un mètode d'asserció en el qual es comprova el resultat esperat de l'execució de codi en comparació del resultat real.

Vegeu amb detall el test unitari del punt 1.3:

```

1  public class DBConnectionTest {
2      DBConnection dBConnection;
3      Connection connection;
4
5      @Before
6      public void setUp(){
7          dBConnection = new DBConnection();
8      }
9
10     @After
11     public void cleanUp() throws SQLException {
12         connection.close();
13     }
14
15     @Test
16     public void conectarAmbLaBaseDeDades() throws IOException, SQLException {
17         connection = dBConnection.getConnection();
18         Assert.assertEquals("H2 JDBC Driver", connection.getMetaData().
19             getDriverName());
20         Assert.assertEquals("SOCIOC_DB", connection.getCatalog());
21     }

```

A la línies 5-9 hi ha el següent codi:

```

1  @Before
2  public void setUp(){
3      dBConnection = new DBConnection();
4

```

L'anotació `@Before` serveix per marcar una funció que s'executarà abans de l'execució de cada test i serveix per assegurar que tots els tests sempre parteixen de les mateixes condicions per ser executats. De manera similar, les línies 10-13:

```

1  @After
2  public void cleanUp() throws SQLException {
3      connection.close();
4

```

Sempre s'executaran en acabar cada un dels tests unitaris. En el vostre cas us assegurareu que en iniciar un test s'establirà connexió amb la base de dades, i en acabar es tancarà la connexió.

1.4.2 Cobertura dels tests unitaris

Una cosa fonamental és saber quina part del vostre codi està cobert pels tests unitaris. Idealment s'ha d'intentar aconseguir un 100%, però normalment es considera com a acceptable una cobertura de 80-90%. La versió 7 de Netbeans ja té incorporat un *plugin* que permet veure la cobertura de projectes Maven (MavenCodeCoverage). Afegiu el següent al final del fitxer pom.xml.

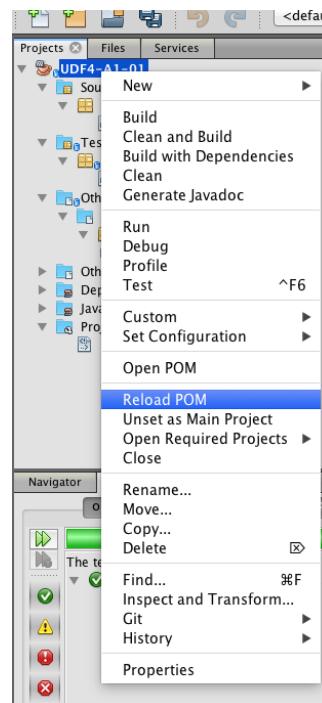
```

1 <plugin>
2   <groupId>org.jacoco</groupId>
3   <artifactId>jacoco-maven-plugin</artifactId>
4   <version>0.7.5.201505241946</version>
5   <executions>
6     <execution>
7       <goals>
8         <goal>prepare-agent</goal>
9       </goals>
10      </execution>
11      <execution>
12        <id>report</id>
13        <phase>prepare-package</phase>
14        <goals>
15          <goal>report</goal>
16        </goals>
17      </execution>
18    </executions>
19  </plugin>

```

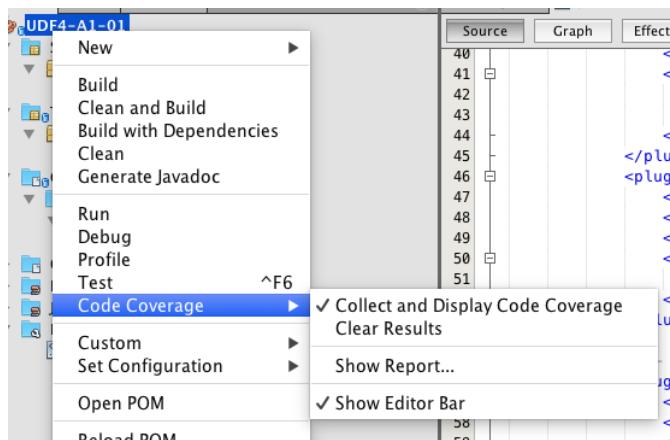
A continuació només fa falta dir a Netbeans que torni a carregar el pom.xml, com podeu veure en la figura 1.10.

FIGURA 1.10. Recarregar pom.xml



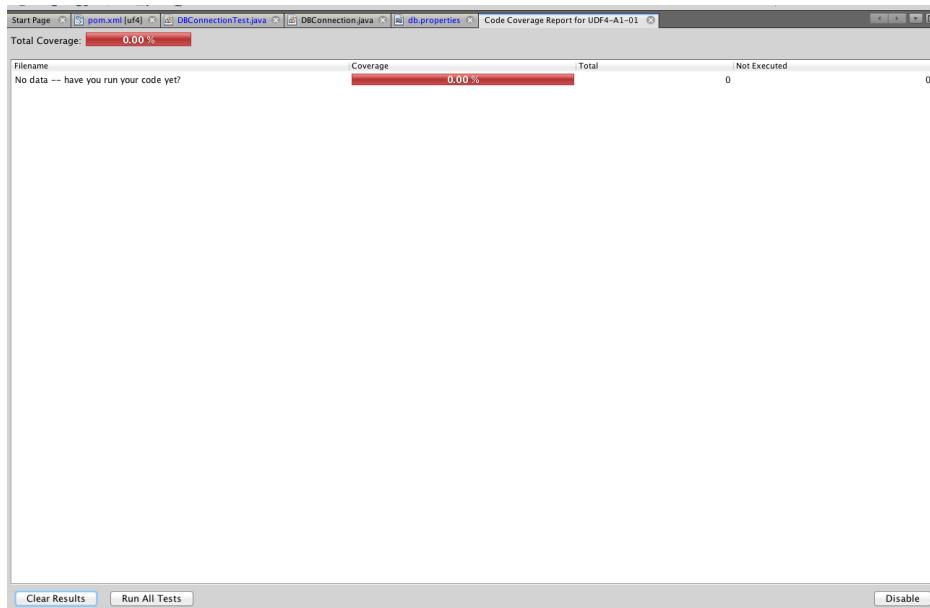
Un cop fet això ja podeu accedir a la cobertura dels tests unitaris que teniu al projecte, tal com podeu veure en la figura 1.11 i seleccionant *Show report*.

FIGURA 1.11. Cobertura



Inicialment no hi ha informació, així que haureu d'executar tots els tests prenent el botó *Run All Tests* (vegeu la figura 1.12).

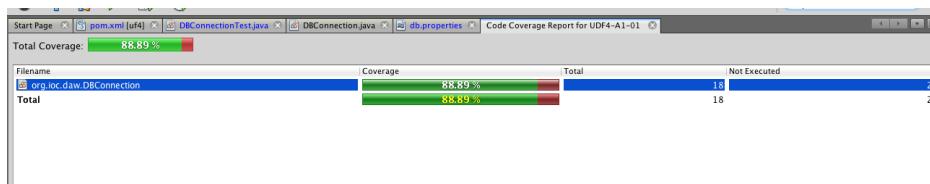
FIGURA 1.12. Report de cobertura inicial



Un cop el vostre test s'ha executat podeu veure que tenim una cobertura d'un 88.89% (vegeu la figura 1.13).

Per veure el resultat heu de tancar i tornar a obrir la pestanya on es mostren els resultats (*Code Coverage report*).

FIGURA 1.13. Report de cobertura inicial



Si feu clic a la classe que hem testejat (DBConnection) podeu veure les parts del codi que estan testejades i les que no (vegeu la figura 1.14).

FIGURA 1.14. Report de cobertura inicial

```

package org.ioc.daw;

import org.h2.jdbc.JdbcConnection;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class DBConnection {
    public Connection getConnection() throws SQLException, IOException {
        Properties props = new Properties();
        InputStream resourceAsStream = null;
        Connection con = null;
        try {
            ClassLoader classLoader = getClass().getClassLoader();
            URL urlResource = classLoader.getResource("db.properties");
            if (urlResource != null) {
                resourceAsStream = urlResource.openStream();
                props.load(resourceAsStream);
                Class.forName(props.getProperty("DB_DRIVER_CLASS"));
                con = DriverManager.getConnection(props.getProperty("DB_URL"),
                    props.getProperty("DB_USERNAME"),
                    props.getProperty("DB_PASSWORD"));
            }
        } catch (IOException | ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        } finally {
            if (resourceAsStream != null) {
                resourceAsStream.close();
            }
        }
        return con;
    }
}

```

Podeu veure que la part de codi sense cobertura dels nostres tests és la que fa referència a com tractar les excepcions.

1.4.3 Augmentant la cobertura del codi

El codi del qual partireu el teniu disponible a l'apartat d'annexos de la unitat.

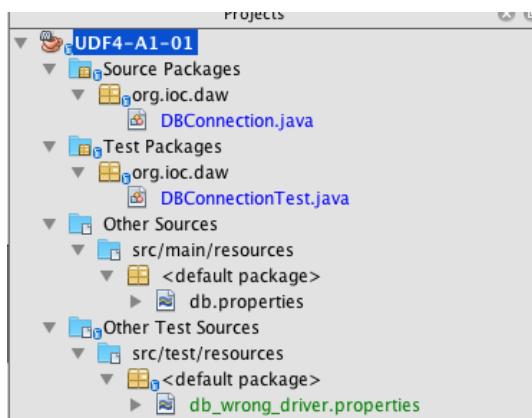
El vostre següent objectiu és aconseguir un 100% de cobertura als tests unitaris. Per aconseguir això heu de canviar el codi, ja que heu de ser capaços de poder crear de manera controlada errors i assegurar-vos que es tracten correctament.

En el vostre cas, una forma fàcil de testejar les excepcions és, per exemple, crear un fitxer de propietats que contingui un nom de *driver* de base de dades incorrecte. Creeu el següent fitxer als recursos de test (figura 1.15) amb el contingut següent i amb el nom db_wrong_driver.properties:

```

1 #BD en memoria H2
2 DB_DRIVER_CLASS=org.h2.WrongDriver
3 DB_URL=jdbc:h2:mem:socioc_db
4 DB_USERNAME=usuari
5 DB_PASSWORD=passws

```

FIGURA 1.15. Fitxer de connexió amb la BD

En aquest exemple, la classe `org.h2.WrongDriver` no existeix, fet que provocarà un error quan s'intenti carregar el *driver* en memòria amb `Class.forName(props.getProperty("DB_DRIVER_CLASS"));`.

Això presenta un problema: a la classe `DBConnection` no hi ha manera de seleccionar quin fitxer de propietats s'ha d'utilitzar. El primer que haureu de fer, doncs, és refactoritzar el mètode `getConnection` perquè accepti un paràmetre que us serveixi per passar a quin fitxer de propietats utilitzar.

Refactorització és el procés de reestructurar el codi d'una aplicació sense canviar la seva funcionalitat per tal de millorar la seva eficiència, estructura, llegibilitat o reutilització.

```

1 public Connection getConnection(String dbProperties) throws SQLException,
2     IOException {
3     Properties props = new Properties();
4     InputStream resourceAsStream = null;
5     Connection con = null;
6     try {
7         ClassLoader classLoader = getClass().getClassLoader();
8         URL urlResource = classLoader.getResource(dbProperties);
9         if (urlResource != null) {
10             resourceAsStream = urlResource.openStream();
11             props.load(resourceAsStream);
12             Class.forName(props.getProperty("DB_DRIVER_CLASS"));
13             con = DriverManager.getConnection(props.getProperty("DB_URL"),
14                 props.getProperty("DB_USERNAME"),
15                 props.getProperty("DB_PASSWORD"));
16         }
17     } catch (IOException | ClassNotFoundException | SQLException e) {
18         e.printStackTrace();
19     } finally {
20         if (resourceAsStream != null) {
21             resourceAsStream.close();
22         }
23     }
24     return con;
}

```

Vegeu aquí un dels avantatges de fer tests unitaris, el fet de pensar com testejar el codi millora el diseny de la nostra API. En aquest cas, afegir aquest paràmetre us permetrà utilitzar el mateix codi amb diferents bases de dades, ja que l'únic

que farà falta serà passar el fitxer de propietats adient. Ara afegireu un test que carregui el fitxer amb la configuració errònia:

```

1 @Test
2 public void dbConnectionWrongDriver() throws IOException, SQLException {
3     connection = dBConnection.getConnection("db_wrong_driver.properties");
4     Assert.assertNull(connection);
5 }
```

Com que l'objecte Connection pot ser ara *null*, s'ha de canviar el que es fa després d'executar el test.

```

1 @After
2 public void cleanUp() throws SQLException {
3     if(connection != null){
4         connection.close();
5     }
6 }
```

Comproveu vosaltres mateixos que ara tenim una cobertura dels tests unitaris del 100%.

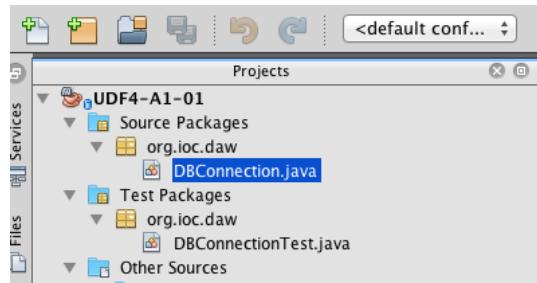
1.5 Fent consultes a la base de dades

A continuació començareu a treballar amb la base de dades que utilitzareu per construir l'aplicació que anomenareu “SocLoc”. La idea general és desenvolupar una xarxa social on els estudiants pugueu preguntar i resoldre preguntes relacionades amb les assignatures que esteu cursant. Les millors respuestas es votaran i anireu guanyant punts i creant-vos una reputació. Començareu per obtenir un llistat dels alumnes que hi ha al sistema.

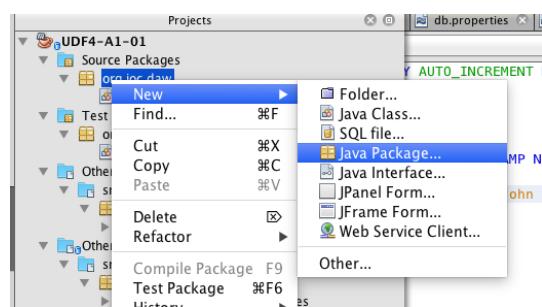
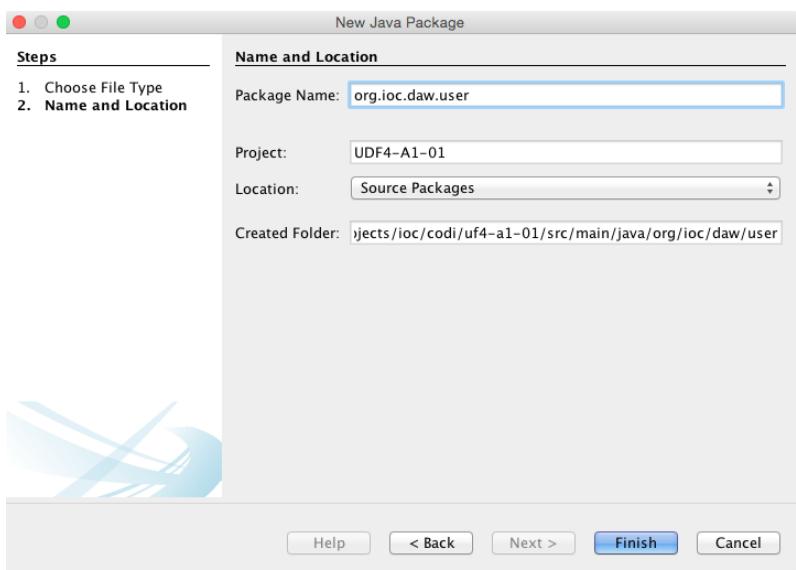
Intenteu contestar a les següents preguntes:

- Quines classes haurem d'afegir al sistema?
- Com obtindrem les dades de la base de dades?

Les bases de dades s'estructuren en taules que tenen una sèrie de registres amb informació. Quan vulgueu extraure una part d'aquesta informació fareu una consulta que us retornarà els registres adients. La vostra aplicació no entén de registres de bases de dades, i necessitareu expressar la informació que voleu guardar a la base de dades com una entitat que pugueu utilitzar a l'aplicació. Llavors, el primer que fareu serà definir una classe que representi un usuari del sistema. Com podeu veure en la figura 1.16, teniu un paquet org.iow.daw que és molt general, així que ara que afegireu més classes és necessari que comenceu a estructurar les classes en diferents paquets.

FIGURA 1.16. Paquet org.ioc.daw

Hi ha moltes formes de fer aquesta organització, i no n'hi cap de correcta o incorrecta mentre siguin consistentes i ben organitzades. La millor estructura és organitzar els paquets en funció de les entitats de domini que hi hagi a la nostra aplicació. Per exemple, el paquet `org.ioc.daw.user` contindrà totes les classes amb funcionalitat relacionada amb els usuaris. Creeu aquest paquet i el paquet `org.ioc.daw.db` tal com es mostra en la figura 1.17 i en la figura 1.18.

FIGURA 1.17. Paquet org.ioc.daw**FIGURA 1.18.** Paquet org.ioc.daw

A partir del codi de partida del present apartat, creeu la classe `User` amb el contingut que es mostra a continuació i moveu la classe `DBConnection` al paquet `org.ioc.daw.db`.

Trobareu el codi de partida per al present apartat als annexos de la unitat.

```

1  public class User {
2      private int userId;
3      private String username;
4      private String name;
5      private String email;
6      private int rank;
7      private Timestamp createdOn;
8      private boolean active;
9
10     public User(int userId, String username, String name, String email, int
11                 rank, Timestamp createdOn, boolean active) {
12         this.userId = userId;
13         this.username = username;
14         this.name = name;
15         this.email = email;
16         this.rank = rank;
17         this.createdOn = createdOn;
18         this.active = active;
19     }
20
21     public int getUserId() {
22         return userId;
23     }
24
25     public String getUsername() {
26         return username;
27     }
28
29     public String getName() {
30         return name;
31     }
32
33     public String getEmail() {
34         return email;
35     }
36
37     public int getRank() {
38         return rank;
39     }
40
41     public Timestamp getCreatedOn() {
42         return createdOn;
43     }
44
45     public boolean isActive() {
46         return active;
47     }

```

Com que aquesta classe no té cap lògica, no fa falta escriure un test unitari. Seguidament definireu una classe que s'encarregui de fer les operacions amb la base de dades. Normalment, a aquest tipus de classes se les anomena *Data Access Objects* (DAO, objectes d'accés a dades), per la qual cosa anomenareu la vostra classe UserDAO.

```

1  public class UserDAO {
2      public List<User> findAllUsers() {
3          String qry = "select user_id, username, name, email, rank, active,
4                      created_on from users";
5          DBConnection dBConnection = new DBConnection();
6          List<User> users = new ArrayList<>();
7          try {
8              Connection conn = dBConnection.getConnection("db.properties");
9              Statement stmt = conn.createStatement();
10             ResultSet rs = stmt.executeQuery(qry); {
11                 while (rs.next()) {

```

```

11     int userId = rs.getInt("user_id");
12     String username = rs.getString("username");
13     String name = rs.getString("name");
14     String email = rs.getString("email");
15     int rank = rs.getInt("rank");
16     boolean active = rs.getBoolean("active");
17     Timestamp timestamp = rs.getTimestamp("created_on");
18     User user = new User(userId, username, name, email, rank,
19                           timestamp, active);
20     users.add(user);
21   }
22 } catch (SQLException | IOException e) {
23   e.printStackTrace();
24 }
25 return users;
26 }
```

Una de les operacions fets més comunament contra una base de dades és una consulta. Fer consultes de bases de dades utilitzant JDBC és bastant fàcil, encara que hi ha una mica de codi repetitiu que s'ha d'utilitzar cada vegada que s'executa una consulta. En primer lloc, es necessita obtenir un objecte de connexió amb la BD. Després es crea una consulta i es guarda a una variable de tipus *string*. La línia 3 defineix la consulta que es farà a la base de dades. En aquesta estem demanant a la BD que retorni tots els registres de la taula “Users”.

```
1 select user_id, username, name, email, rank, active, created_on from users;
```

A continuació s'utilitza una clàusula *try-with-resources* per crear els objectes que són necessaris per fer la consulta de la base de dades. Això farà que si hi ha qualsevol problema i el programa llença una excepció, tots els recursos es tancaran automàticament.

```

1 try (
2   Connection conn = dBConnection.getConnection("db.properties");
3   Statement stmt = conn.createStatement();
4   ResultSet rs = stmt.executeQuery(qry);
```

En aquest cas, si hi ha cap problema creant la connexió amb la BD, creant o executant la consulta, tancarem automàticament la connexió amb la BD.

Com es pot veure, el mètode *executeQuery* accepta un *string* i retorna un objecte *ResultSet*. L'objecte *ResultSet* fa que sigui fàcil treballar amb els resultats de la consulta. Si observeu la següent línia de codi (línia 9), un bucle *while* es crea amb *rs.next()*. Aquest bucle recorrerà els continguts de l'objecte *ResultSet*, obtenint la següent fila que es retorna des de la consulta amb cada iteració. Una vegada totes les files retornades han estat processades, *rs.next()* retornarà *false* per indicar que no hi ha més resultats que processar.

Dins del bucle *while*, l'objecte *ResultSet* s'utilitza per obtenir els valors dels noms de les columnes indicades amb cada passada. Observeu que si s'espera que la columna retorna un *string* heu d'utilitzar el mètode *ResultSet.getString*, passant el nom de la columna en format *string*. De la mateixa manera, si s'espera que la columna retorna un *int* haureu d'utilitzar el mètode *ResultSet.getInt*. Un cop tenim tota la informació necessària, es creen els objectes de tipus *User* i s'afegeixen a la llista de tots els usuaris (línies 18-19).

A continuació necessiteu fer un test unitari per assegurar-vos que la classe UserDAO té el funcionament desitjat. Teniu un problema, però: el fitxer db.properties conté les dades d'accés a la BD principal o de producció. Per fer els tests voleu poder configurar la BD que utilitzareu, ja que això permetrà usar una BD específica per executar-los. Refactoritzareu el codi per tal que l'objecte DBConnection no es creï dintre de la classe UserDAO, sinó que es passi com a paràmetre. D'aquesta forma aconseguireu el desacoblament de la classe que fa consultes a la BD amb la qual s'encarrega de fer la connexió.

```

1  public class UserDAO {
2      private DBConnection dBConnection;
3      public UserDAO(DBConnection dBConnection){
4          this.dBConnection = dBConnection;
5      }
6
7      public List<User> findAllUsers() {
8          String qry = "select user_id, username, name, email, rank, active,
9              created_on from users";
10         List<User> users = new ArrayList<>();
11         try {
12             Connection conn = getDBConnection().getConnecion();
13             Statement stmt = conn.createStatement();
14             ResultSet rs = stmt.executeQuery(qry);
15             while (rs.next()) {
16                 int userId = rs.getInt("user_id");
17                 String username = rs.getString("username");
18                 String name = rs.getString("name");
19                 String email = rs.getString("email");
20                 int rank = rs.getInt("rank");
21                 boolean active = rs.getBoolean("active");
22                 Timestamp timestamp = rs.getTimestamp("created_on");
23                 User user = new User(userId, username, name, email, rank,
24                     timestamp, active);
25                 users.add(user);
26             }
27         } catch (SQLException | IOException e) {
28             e.printStackTrace();
29         }
30         return users;
31     }
32
33     public DBConnection getDBConnection(){
34         return this.dBConnection;
35     }
36 }
```

Una forma de fer això és declarar DBConnection dBConnection com un atribut de la classe UserDAO i fer que sigui necessari per a la creació de l'objecte.

```

1  public class UserDAO {
2      private DBConnection dBConnection;
3      public UserDAO(DBConnection dBConnection){
4          this.dBConnection = dBConnection;
5      }
5 }
```

A la classe DBConnection, el fitxer amb els paràmetres de connexió amb la base de dades és imprescindible, així que també refactoritzareu aquesta classe.

```

1  public class DBConnection {
2      private String connectionFile;
3      public DBConnection(String connectionFile) {
4          this.connectionFile = connectionFile;
5      }
5 }
```

A més a la classe DBConnection el mètode getConnection ha d'anar sense paràmetres perquè s'ha de fer servir connectionFile.

Aquest últim canvi trencarà el test unitari, així que haureu de canviar la classe DBConnectionTest.

Ara ja podeu crear el test unitari per testejar la classe UserDao.

```

1 public class UserDaoTest {
2     private DBConnection dBConnection;
3     private String connectionProperties = "db-test.properties";
4     @Before
5     public void setUp(){
6         dBConnection = new DBConnection(connectionProperties);
7     }
8
9     @Test
10    public void findAllUsers(){
11        UserDao userDao = new UserDao(dBConnection);
12        List<User> users = userDao.findAllUsers();
13        Assert.assertEquals("Hauriem de tenir 2 usuaris a la base de dades", 2,
14                           users.size());
15    }
}

```

Abans de poder executar el test, creareu el fitxer de connexió amb la BD de test.

```

1 DB_DRIVER_CLASS=org.h2.Driver
2 DB_URL=jdbc:h2:mem:socioc_db;INIT=runscript from 'classpath:init.sql';
3 DB_USERNAME=usuari
4 DB_PASSWORD=passwd

```

Com que en aquest cas és una BD en memòria, no necessiteu canviar el nom de la BD. La part més important d'aquest fitxer és la segona línia, que executarà una sèrie d'instruccions SQL que asseguraran que la BD de test sempre estarà en el mateix estat. En la figura 1.19 es mostren els fitxers que heu de tenir abans d'executar els tests unitaris.

FIGURA 1.19. Fitxers pel test unitari de UserDao



```

1  CREATE TABLE users(user_id INT PRIMARY KEY AUTO_INCREMENT NOT NULL,
2                      username VARCHAR(10) NOT NULL,
3                      name VARCHAR(20) NOT NULL,
4                      email VARCHAR(50) NOT NULL,
5                      rank INT DEFAULT 0,
6                      active BOOLEAN DEFAULT true,
7                      created_on TIMESTAMP AS CURRENT_TIMESTAMP NOT NULL);
8
9  INSERT INTO users (username, name, email) VALUES ('user1', 'John Test', '
10                         john@email.com');
11  INSERT INTO users (username, name, email) VALUES ('user2', 'Paul Test', '
12                         paul@email.com');

```

1.6 Operacions CRUD

CRUD correspon a l'acrònim *Create Read Update Delete*: crear, recuperar , actualitzar i eliminar, i es refereix a les quatre funcions principals que implementareu quan desenvolupeu aplicacions de bases de dades. Les funcions CRUD són les que donen la capacitat a les aplicacions de comportar-se d'una manera dinàmica, ja que les dades podran ser canviades pels usuaris. Aquests podran crear, visualitzar, modificar i alterar les dades. Les operacions CRUD permeten accedir i manipular les entitats definides a les bases de dades.

Per exemple, a la taula d'usuaris definida a l'apartat 1.5, *Create* implicarà afegir un nou usuari; *Read*, accedir a les dades d'un o diversos estudiants; *Update* modificarà les dades dels usuaris i *Delete* eliminarà un o diversos registres de la base de dades.

El primer que farem serà modificar la classe UserDAO per permetre recuperar les dades de la base de dades. Recordeu que l'objectiu és construir una aplicació que permeti preguntar i resoldre preguntes relacionades amb les assignatures que esteu cursant. Les millors respuestes es votaran i anireu guanyant punts i creant-vos una reputació. Quines creieu, doncs, que seran el tipus d'informació que necessitareu extreure de la base de dades? Intenteu contestar a les següents preguntes:

- En quines situacions necessitareu obtenir la informació d'un usuari?
- Per mostrar un llistat amb els alumnes que mostri un *ranking* dels alumnes més ben classificats, quina informació necessitareu?

Les operacions de lectura que haureu de dissenyar per a la vostra aplicació estaran totalment determinades per la seva funcionalitat. Tot i així, veureu amb la vostra experiència que hi haurà unes operacions que es repetiran i seran comuns per a moltes aplicacions. Per exemple, en el cas que us ocupa, una aplicació amb usuaris que es poden registrar i donar-se de baixa, moltes de les operacions que definim serviran per a la majoria d'aplicacions que us trobareu, des d'una botiga amb comerç electrònic, un joc *online* o una xarxa social.

1.6.1 Operacions de lectura

Primer de tot, quan un usuari es registri en el sistema haurà d'introduir el seu nom, el nom d'usuari que desitja i el seu correu electrònic. Què passa si el nom d'usuari ja està en ús per part d'un altre usuari? I el correu electrònic? En aquests casos hauríem d'avalar l'usuari que ha d'escollar un altre nom d'usuari o correu electrònic. Ja tenim dues operacions de lectura que haurem d'implementar: trobar usuaris a partir del correu electrònic i a partir del nom d'usuari. Implementem-les:

```

1 public User findUserByEmail(String userEmail){
2     String qry = "select * from users where email ='" + userEmail +"'";
3
4     User user = null;
5     try {
6         Connection conn = dBConnection.getConnection();
7         Statement stmt = conn.createStatement();
8         ResultSet rs = stmt.executeQuery(qry);) {
9             while (rs.next()) {
10                 int userId = rs.getInt("user_id");
11                 String username = rs.getString("username");
12                 String name = rs.getString("name");
13                 String email = rs.getString("email");
14                 int rank = rs.getInt("rank");
15                 boolean active = rs.getBoolean("active");
16                 Timestamp timestamp = rs.getTimestamp("created_on");
17                 user = new User(userId, username, name, email, rank, timestamp, active);
18             }
19         } catch (SQLException | IOException e) {
20             e.printStackTrace();
21         }
22     return user;

```

La part més important és on es defineix la consulta que es farà a la base de dades:

```

1 String qry = "select * from users where email ='" + userEmail +"'";

```

Aquesta és la consulta SQL que retornarà un usuari a partir del seu correu electrònic. Si no hi ha cap usuari registrat amb el correu electrònic retornarà *null*. Això es pot expressar amb el següent test unitari:

```

1 @Test
2 public void findUserByEmail(){
3     String existingEmail = "john@email.com";
4     String unknownEmail = "does.not@exist.com";
5
6     UserDAO userDAO = new UserDAO(dBConnection);
7     User user = userDAO.findUserByEmail(existingEmail);
8     Assert.assertNotNull(user);
9     user = userDAO.findUserByEmail(unknownEmail);
10    Assert.assertNull(user);
11 }

```

`Assert.assertNotNull` verifica que un objecte no és *null*, és a dir, s'ha trobat un resultat a la base de dades. Al contrari, `Assert.assertNull` verifica que un objecte és *null*; en el vostre cas, que no s'ha trobat cap usuari a la base de dades amb un cert correu electrònic.

A continuació implementareu un mètode que us permeti trobar usuaris a partir del nom d'usuari.

```

1  public User findUserByUsername(String username){
2      String qry = "select * from users where username ='" + username +"'";
3      User user = null;
4      try {
5          Connection conn = dBConnection.getConnection();
6          Statement stmt = conn.createStatement();
7          ResultSet rs = stmt.executeQuery(qry);) {
8              while (rs.next()) {
9                  int userId = rs.getInt("user_id");
10                 username = rs.getString("username");
11                 String name = rs.getString("name");
12                 String email = rs.getString("email");
13                 int rank = rs.getInt("rank");
14                 boolean active = rs.getBoolean("active");
15                 Timestamp timestamp = rs.getTimestamp("created_on");
16                 user = new User(userId, username, name, email, rank, timestamp,
17                                 active);
18             }
19         } catch (SQLException | IOException e) {
20             e.printStackTrace();
21         }
22     return user;
}

```

I el test unitari:

```

1  @Test
2  public void findUserByUsername(){
3      String existingUsername = "user1";
4      String unknownUsername = "unknown";
5
6      UserDAO userDAO = new UserDAO(dBConnection);
7      User user = userDAO.findUserByUsername(existingUsername);
8      Assert.assertNotNull(user);
9      user = userDAO.findUserByUsername(unknownUsername);
10     Assert.assertNull(user);
11 }

```

Doneu una ullada a la classe sencera, ja que hi ha codi repetit; símptoma inequívoc que és hora de refactoritzar el codi.

```

1  public class UserDAO {
2      private DBConnection dBConnection;
3      public UserDAO(DBConnection dBConnection){
4          this.dBConnection = dBConnection;
5      }
6
7      public List<User> findAllUsers() {
8          String qry = "select user_id, username, name, email, rank, active,
9                      created_on from users";
10         List<User> users = new ArrayList<>();
11         try {
12             Connection conn = dBConnection.getConnection();
13             Statement stmt = conn.createStatement();
14             ResultSet rs = stmt.executeQuery(qry);) {
15                 while (rs.next()) {
16                     int userId = rs.getInt("user_id");
17                     String username = rs.getString("username");
18                     String name = rs.getString("name");
19                     String email = rs.getString("email");
20                     int rank = rs.getInt("rank");
21                     boolean active = rs.getBoolean("active");
22                     Timestamp timestamp = rs.getTimestamp("created_on");
23                     User user = new User(userId, username, name, email, rank, timestamp,
24                                         active);
25                     users.add(user);
26                 }
27             }
28         } catch (SQLException e) {
29             e.printStackTrace();
30         }
31     return users;
32 }

```

```

22         User user = new User(userId, username, name, email, rank, timestamp,
23             active);
24         users.add(user);
25     }
26     } catch (SQLException | IOException e) {
27         e.printStackTrace();
28     }
29     return users;
30 }
31
32 public User findUserByEmail(String userEmail){
33     String qry = "select * from users where email ='" + userEmail +"'";
34     User user = null;
35     try {
36         Connection conn = dBConnection.getConnection();
37         Statement stmt = conn.createStatement();
38         ResultSet rs = stmt.executeQuery(qry);) {
39         while (rs.next()) {
40             int userId = rs.getInt("user_id");
41             String username = rs.getString("username");
42             String name = rs.getString("name");
43             String email = rs.getString("email");
44             int rank = rs.getInt("rank");
45             boolean active = rs.getBoolean("active");
46             Timestamp timestamp = rs.getTimestamp("created_on");
47             user = new User(userId, username, name, email, rank, timestamp,
48                 active);
49         }
50     } catch (SQLException | IOException e) {
51         e.printStackTrace();
52     }
53     return user;
54 }
55
56 public User findUserByUsername(String username){
57     String qry = "select * from users where username ='" + username +"'";
58     User user = null;
59     try {
60         Connection conn = dBConnection.getConnection();
61         Statement stmt = conn.createStatement();
62         ResultSet rs = stmt.executeQuery(qry);) {
63         while (rs.next()) {
64             int userId = rs.getInt("user_id");
65             username = rs.getString("username");
66             String name = rs.getString("name");
67             String email = rs.getString("email");
68             int rank = rs.getInt("rank");
69             boolean active = rs.getBoolean("active");
70             Timestamp timestamp = rs.getTimestamp("created_on");
71             user = new User(userId, username, name, email, rank, timestamp,
72                 active);
73         }
74     } catch (SQLException | IOException e) {
75         e.printStackTrace();
76     }
77     return user;
78 }
79 }
```

Si us hi fixeu, la part del codi que s'encarrega de fer la connexió amb la base de dades i extreure els resultats es repeteix, millor posar-la a un mètode. Per què? Imagineu la següent situació: més endavant, durant el desenvolupament de l'aplicació, volem afegir un altre atribut a la classe User, com per exemple la data de naixement. Tal com tenim el codi ara mateix, fer aquest canvi implicarà canviar tres mètodes: `findAllUsers`, `findUserByEmail` i `findUserByUsername`. Si

aquest codi comú el poseu en un mètode separat, el canvi només l'haureu de fer en un lloc.

```

1  public class UserDAO {
2      private DBConnection dBConnection;
3      public UserDAO(DBConnection dBConnection){
4          this.dBConnection = dBConnection;
5      }
6
7      public List<User> findAllUsers() {
8          String qry = "select user_id, username, name, email, rank, active,
9              created_on from users";
10         List<User> users = new ArrayList<>();
11         try (
12             Connection conn = dBConnection.getConnection();
13             Statement stmt = conn.createStatement();
14             ResultSet rs = stmt.executeQuery(qry)); {
15             while (rs.next()) {
16                 User user = buildUserFromResultSet(rs);
17                 users.add(user);
18             }
19         } catch (SQLException | IOException e) {
20             e.printStackTrace();
21         }
22         return users;
23     }
24
25     public User findUserByEmail(String userEmail){
26         String qry = "select * from users where email ='" + userEmail + "'";
27         User user = null;
28         try (
29             Connection conn = dBConnection.getConnection();
30             Statement stmt = conn.createStatement();
31             ResultSet rs = stmt.executeQuery(qry)); {
32             while (rs.next()) {
33                 user = buildUserFromResultSet(rs);
34             }
35         } catch (SQLException | IOException e) {
36             e.printStackTrace();
37         }
38         return user;
39     }
40
41     public User findUserByUsername(String username){
42         String qry = "select * from users where username ='" + username + "'";
43         User user = null;
44         try (
45             Connection conn = dBConnection.getConnection();
46             Statement stmt = conn.createStatement();
47             ResultSet rs = stmt.executeQuery(qry)); {
48             while (rs.next()) {
49                 user = buildUserFromResultSet(rs);
50             }
51         } catch (SQLException | IOException e) {
52             e.printStackTrace();
53         }
54         return user;
55     }
56
57     private User buildUserFromResultSet(ResultSet rs) throws SQLException{
58         int userId = rs.getInt("user_id");
59         String username = rs.getString("username");
60         String name = rs.getString("name");
61         String email = rs.getString("email");
62         int rank = rs.getInt("rank");
63         boolean active = rs.getBoolean("active");
64         Timestamp timestamp = rs.getTimestamp("created_on");
65         User user = new User(userId, username, name, email, rank, timestamp,
66             active);
67         return user;
68     }

```

```
66     }
67 }
```

Hem creat la funció `buildUserFromResultSet`, que s'encarrega de construir un objecte usuari a partir del resultat d'executar la consulta SQL. Un dels avantatges de tenir tests unitaris és que podeu assegurar-vos que la refactorització no ha trencat la funcionalitat del vostre codi. Si correu els tests a la classe `UserDAOTest` ha de continuar passant.

Encara es pot refactoritzar més, ja que totes les funcions tenen la mateixa estructura:

- Establir connexió amb la base de dades
- Executar una consulta
- Iterar sobre resultats i construir un o diversos objectes User

L'única diferència és que hi haurà funcions que retornaran només un usuari o diversos. Això ho podem solucionar creant les següents funcions:

```
1 private User findUniqueResult(String query) throws Exception{
2     List<User> users = executeQuery(query);
3     if(users.isEmpty()){
4         return null;
5     }
6     if(users.size() > 1){
7         throw new Exception("Only one result expected");
8     }
9     return users.get(0);
10 }
11
12 private List<User> executeQuery(String query){
13     List<User> users = new ArrayList<>();
14     try {
15         Connection conn = dBConnection.getConnection();
16         Statement stmt = conn.createStatement();
17         ResultSet rs = stmt.executeQuery(query); {
18             while (rs.next()) {
19                 User user = buildUserFromResultSet(rs);
20                 if(user != null){
21                     users.add(user);
22                 }
23             }
24         } catch (SQLException | IOException e) {
25             e.printStackTrace();
26         }
27         return users;
28     }
```

La funció `executeQuery` s'encarrega d'establir la connexió amb la base de dades i retornar tots els usuaris que resultin d'executar la consulta. En aquest cas no sabem si retornarà un, diversos o cap resultat. Això es pot expressar amb el tipus de dades `List`; si la llista està buida voldrà dir que no hi ha cap resultat, i si n'hi ha, n'hi haurà tants com elements contingui la llista. Per altra part, la funció `findUniqueResult` examinarà la llista retornada per `executeQuery` i retornarà un objecte `null` si no hi ha cap resultat per a la consulta; un objecte `User`, si hi és, troba un resultat i es llençarà una excepció si hi ha un error. Per avisar la vostra aplicació que alguna cosa ha anat malament llençareu una excepció.

```

1 if(users.size() > 1){
2     throw new Exception("Only one result expected");
3 }
```

Aquesta excepció la propagareu a capes superiors de l'aplicació, on decidireu què fer. Les funcions que permeten obtenir informació sobre els usuaris quedaran molt més simplificades.

```

1 public List<User> findAllUsers() {
2     String qry = "select user_id, username, name, email, rank, active,
3                 created_on from users";
4     List<User> users = executeQuery(qry);
5     return users;
6 }
7 public User findUserByEmail(String userEmail) throws Exception{
8     String qry = "select * from users where email ='"+userEmail+"'";
9     return findUniqueResult(qry);
10 }
11 public User findUserByUsername(String username) throws Exception{
12     String qry = "select * from users where username ='"+username+"'";
13     return findUniqueResult(qry);
14 }
15 }
```

1.6.2 Operacions d'escriptura

Hi ha dues formes diferents d'escriure dades: crear una nova entrada a la base de dades o actualitzar un registre existent.

Comencareu a treballar a partir del codi que teniu disponible a l'apartat d'annexos de la unitat.

Contesteu a les següents preguntes:

- Quines operacions necessitareu de creació de dades?
- Quines faran falta d'actualització?

Com podeu veure al codi, en aquest moment només heu definit els usuaris, llavors només podreu crear usuaris. Per altra part, respecte a l'actualització hi ha diverses operacions que caldrà fer:

- Crear un usuari.
- Canviar el correu electrònic.
- Actualitzar el *ranking*.
- Actualitzar el nom.

La creació es fa mitjançant l'operació SQL *INSERT* i l'actualització amb *UPDATE*. Voleu crear un usuari a partir del nom, nom d'usuari i correu electrònic. Què passarà amb la resta d'atributs de l'objecte User (*userId*, *rank*, *createdOn* i *active*)? Les principals diferències amb el codi per llegir dades seran dues: la

sentència SQL, que ara contindrà la paraula clau *INSERT*, i l'operació que cridarem en l'objecte Statement, que serà executeUpdate en lloc de executeQuery.

```

1 public User createUser(String username, String name, String email) throws
2     Exception {
3     String qry = "INSERT INTO users (username, name, email) VALUES ('"
4         + username + "', '"'
5         + name + "', '"'
6         + email + "'"
7         + ");";
8     try {
9         Connection conn = dBConnection.getConnection();
10        Statement stmt = conn.createStatement();
11
12        int result = stmt.executeUpdate(qry);
13
14        return findUserByUsername(username);
15    } catch (SQLException | IOException e) {
16        e.printStackTrace();
17        return null;
18    }
19 }
```

El mètode executeUpdate (línia 11) retornarà la quantitat de registres que s'han inserit en la base de dades. En alguns entorns, el resultat d'un insert pot retornar 0, per aquest motiu sempre s'ha de controlar l'èxit de l'operació amb excepcions. Finalment (línia 15), un cop creat el nou usuari voldreu retornar l'objecte User amb tots els seus atributs. Aproveiteu la funció findUserByUsername per recuperar l'usuari recentment creat de la base de dades. A continuació podeu veure el test corresponent:

```

1 @Test
2 public void createUser() throws Exception {
3     String username = "testUser";
4     String name = "Pete Test";
5     String email = "pete@email.com";
6     UserDAO userDAO = new UserDAO(dBConnection);
7     User createdUser = userDAO.createUser(username, name, email);
8     Assert.assertNotNull(createdUser);
9     Assert.assertEquals(username, createdUser.getUsername());
10    Assert.assertNotEquals(0, createdUser.getUserId());
11 }
```

Primer comproveu que l'usuari creat no és *null*, que el nom d'usuari correspon al nom que hem passat a la funció createUser i que l'atribut userId del nou usuari no és 0.

Quan es va definir la taula, el camp “user_id” es va definir com *user_id INT PRIMARY KEY AUTO_INCREMENT NOT NULL*; llavors, si el nou usuari s’ha afegit correctament a la base de dades, es pot afirmar que el seu valor no serà *null*.

Quan executeu aquest test us trobareu que hi ha un error:

```

1 org.h2.jdbc.JdbcSQLException: Table "USERS" already exists; SQL statement:
2 CREATE TABLE users(user_id INT PRIMARY KEY AUTO_INCREMENT NOT NULL,
3                     username VARCHAR(10) NOT NULL,
4                     name   VARCHAR(20) NOT NULL,
```

```

5           email VARCHAR(50) NOT NULL,
6           rank INT DEFAULT 0,
7           active BOOLEAN DEFAULT true,
8           created_on TIMESTAMP AS CURRENT_TIMESTAMP NOT NULL)
9           [42101-190]
at org.h2.message.DbException.getJdbcSQLException(DbException.java:345)

```

Què ha passat? A la funció `createUser`, després de definir la consulta a la base de dades, el que feu és establir una connexió amb la base de dades:

```

1 Connection conn = dBConnection.getConnection();

```

Quan després crideu la funció `findUserByUsername`, s'executarà la mateixa funció. Recordeu que aquest mètode crea una connexió amb la base de dades de la següent manera:

```

1 con = DriverManager.getConnection(props.getProperty("DB_URL"),
2                               props.getProperty("DB_USERNAME"),
3                               props.getProperty("DB_PASSWORD"));

```

Aquestes propietats estan definides al fitxer de propietats de la base dades, en el cas dels tests (`db_test.properties`):

```

1 DB_URL=jdbc:h2:mem:socio_db;INIT=runscript from 'classpath:init.sql';
2 DB_USERNAME=usuari
3 DB_PASSWORD=passwd

```

La primera vegada que s'executa `dBConnection.getConnection()` s'executarà el fitxer `init.sql`, que crearà la taula “Users”. Afegireu l'usuari a la base de dades, i en executar `findUserByUsername` s'intentarà executar de nou `init.sql`, però com que ja està creada l'aplicació llençarà aquest error.

Això posa de rellevància un problema del vostre codi: no s'estan reutilitzant les connexions amb la base de dades. Fer una connexió amb la bases de dades és una operació molt cara en termes de recursos: primer s'ha de fer la connexió a través de la xarxa amb la base de dades; s'ha d'inicialitzar una sessió de connexió, que sovint requereix molt de temps de processament per fer l'autenticació d'usuari, establir contextos transaccionals i definir altres aspectes de la sessió que es requereixen per a l'ús de bases de dades subsegüent. En properes seccions veurem com solucionar aquest problema utilitzant *pools* de connexió. La idea és crear una sèrie de connexions amb la base de dades a l'inici de l'aplicació que es compartiran per a totes les operacions que es facin. De moment, però, refactoritzarem el nostre codi de `UserDAO` per permetre reutilitzar les connexions establertes amb la base de dades. Això implica afegir una nova propietat i modificar el mètode `executeQuery`.

```

1 public class UserDAO {
2     private DBConnection dBConnection;
3     private Connection connection;
4
5     private List<User> executeQuery(String query) {
6         List<User> users = new ArrayList<>();
7
8         if (getConnection() == null) {
9             try {

```

```

10         setConnection(dBConnection.getConnection());
11     } catch (SQLException | IOException e) {
12         e.printStackTrace();
13     }
14 }
15 try {
16     Statement stmt = getConnection().createStatement();
17     ResultSet rs = stmt.executeQuery(query)) {
18     while (rs.next()) {
19         User user = buildUserFromResultSet(rs);
20         users.add(user);
21     }
22 } catch (SQLException e) {
23     e.printStackTrace();
24 }
25 return users;
26 }
27
28 public Connection getConnection() {
29     return connection;
30 }
31
32 public void setConnection(Connection connection) {
33     this.connection = connection;
34 }
35
36 }
```

El que heu fet per solucionar el problema és afegir un atribut a la classe UserDAO i, abans de crear-ne un de nou, comprovar si hi ha objecte Connection vàlid; si és el cas, el reutilitzeu. Abans d'executar el test heu d'assegurar-vos que tanquem la connexió amb la base de dades per tal de a cada test hi hagi només les dades definides al fitxer init.sql. Un cop executat, us assegurareu que la connexió amb la base de dades es tanca de manera que cada vegada la base de dades es torni a crear de nou.

```

1 @Before
2 public void setUp() {
3     dBConnection = new DBConnection(connectionProperties);
4     userDAO = new UserDAO(dBConnection);
5 }
6 @After
7 public void tearDown() throws IOException, SQLException {
8     userDAO.getConnection().close();
9 }
```

Afegiu ara un altre test per comprovar què passaria en cas que hi hagi un error creant l'usuari. En aquest cas provoquem l'error introduint un nom d'usuari incorrecte:

```

1 @Test(expected = Exception.class)
2 public void createUserWithError() throws Exception {
3     String username = "sl','sls";
4     String name = "Pete Test";
5     String email = "pete@email.com";
6     User createdUser = userDAO.createUser(username, name, email);
7     Assert.assertNull(createdUser);
8 }
```

Per actualitzar un registre, l'únic que canvia respecte a l'escriptura és que s'utilitza la clàusula SQL *UPDATE* enlloc d'*INSERT*. Vegeu com podeu modificar el correu

electrònic d'un usuari: afegireu la funció `executeUpdateQuery` a `UserDAO` i escriureu el corresponent test unitari.

```

1  private int executeUpdateQuery(String query) {
2      int result = 0;
3      if (getConnection() == null) {
4          try {
5              setConnection(dBConnection.getConnection());
6          } catch (SQLException | IOException e) {
7              e.printStackTrace();
8          }
9      }
10     try {
11         Statement stmt = getConnection().createStatement();
12         result = stmt.executeUpdate(query);
13     } catch (SQLException e) {
14         e.printStackTrace();
15     }
16     return result;
17 }
18 </java>
19
20 <code java>
21 public User updateUserEmail(String username, String newEmail) throws Exception
22 {
23     String qry = "UPDATE users "
24         + "SET email = '" + newEmail + "' "
25         + "WHERE username = '" + username + "' "
26         + ";";
27     int result = executeUpdateQuery(qry);
28
29     return findUserByUsername(username);
}

```

I el test: primer creareu un usuari, modificareu el correu electrònic i comprobareu que és l'única informació que ha canviat:

```

1  @Test
2  public void updateUserEmail() throws Exception {
3      String username = "testUser";
4      String name = "Pete Test";
5      String email = "pete@email.com";
6      User createdUser = userDAO.createUser(username, name, email);
7      Assert.assertNotNull(createdUser);
8      Assert.assertEquals(email, createdUser.getEmail());
9      User updatedUser = userDAO.updateUserEmail(createdUser.getUsername(), "
10         new@email.com");
11     Assert.assertEquals(createdUser.getUserId(), updatedUser.getUserId());
12     Assert.assertEquals("new@email.com", updatedUser.getEmail());
}

```

Si doneu una ullada a les funcions `updateUserEmail` i `createUser` veureu que són molt similars; l'únic que canvia és la consulta a la base de dades. Això vol dir que aquestes dues funcions són candidates per a la refactorització.

```

1  public User createUser(String username, String name, String email) throws
Exception {
2      String qry = "INSERT INTO users (username, name, email) VALUES ('"
3          + username + "', '"'
4          + name + "', ''"
5          + email + "'"
6          + ");";
7      return createOrUpdateUser(username, qry);
8  }
9

```

```

10 public User updateUserEmail(String username, String newEmail) throws Exception
11     {
12         String qry = "UPDATE users "
13             + "SET email = '" + newEmail + "' "
14             + "WHERE username = '" + username + "' "
15             + ";";
16         return createOrUpdateUser(username, qry);
17     }
18
19 private User createOrUpdateUser(String username, String query) throws Exception
20     {
21         int result = executeUpdateQuery(query);
22
23         return findUserByUsername(username);
24     }

```

1.6.3 Eliminació de dades

L'eliminació de dades també es considera una actualització de la base de dades; en aquest cas, modificar-la per eliminar informació.

```

1 public void deleteUser(User user) throws Exception {
2     String query = "DELETE FROM users WHERE user_id = '" + user.getUserId() + " '
3         ";
4     createOrUpdateUser(user.getUsername(), query);
5 }

```

A continuació creeu el test:

```

1 @Test
2 public void deleteUser() throws Exception {
3     String username = "testUser";
4     String name = "Pete Test";
5     String email = "pete@email.com";
6     User createdUser = userDao.createUser(username, name, email);
7     Assert.assertNotNull(createdUser);
8     userDao.deleteUser(createdUser);
9     User deletedUser = userDao.findUserByUsername(username);
10    Assert.assertNull(deletedUser);
11 }

```

La modificació de l'actualització de la base de dades per eliminar la informació es farà utilitzant el mètode `executeUpdate` de l'objecte `Statement`, per a la qual cosa es pot reutilitzar la funció `createOrUpdateUser` que heu creat en l'àpartat "Operacions d'escriptura" i que podeu trobar a l'àpartat d'annexos de la unitat.

1.7 Injecció SQL

Un atac d'injecció SQL és exactament el que el seu nom indica: és quan algú intenta "injectar" el seu codi SQL maliciós a la base de dades d'una altra persona, obligant aquesta base de dades a executar SQL que no estava previst. Això podria arruïnar les seves taules de bases de dades i fins i tot extreure informació valuosa o privada. Vegem-ne un exemple: aquesta és la consulta que hem utilitzat per trobar un usuari a partir del correu electrònic:

```

1 String qry = "select * from users where email ='" + userEmail + "'";

```

És a dir, que per al correu electrònic test@email.com utilitzaréu la crida a la funció `findUserByEmail("test@email.com")`, amb la qual cosa la consulta SQL quedaría:

```
1 select * from users where email ='test@email.com';
```

Un atac d'injecció SQL consistiria a modificar aquest SQL. Com es pot fer? N'hi hauria prou amb modificar l'argument que s'utilitza en cridar la funció. Si executem `findUserByEmail("test@email.com' OR 1=1")`, la consulta SQL que s'executaría seria:

```
1 select * from users where email ='test@email.com' OR '1'='1';
```

La segona condició sempre es complirà; per tant, en lloc d'efectuar una consulta que retorna els usuaris amb el correu, retornarà un llistat de tots els usuaris. En aquest cas estaríeu exposant dades sense voler, però el codi SQL inserit podria ser fàcilment modificat per esborrar totes les dades.

Seguint l'exemple de trobar un usuari a partir del correu electrònic:

```
1 PreparedStatement stmt = connection.prepareStatement("select * from users where
      email =?");
2     stmt.setString(1, userEmail);
```

Per poder utilitzar `PreparedStatement`s refactoritzem el codi per tal de que `executeQuery`, `executeUpdateQuery` i `findUniqueResult` no usin més una cadena de caràcters com a paràmetre. Fixeu-vos també que he introduït un nou mètode `getPreparedStatement` que s'encarrega d'obtenir un objecte `PreparedStatement` ja sigui reutilitzant una connexió existent o creant-ne una de nova:

```
1 public class UserDAO {
2     private DBConnection dBConnection;
3     private Connection connection;
4
5     public UserDAO(DBConnection dBConnection) {
6         this.dBConnection = dBConnection;
7     }
8
9     public List<User> findAllUsers() throws SQLException {
10         String qry = "select user_id, username, name, email, rank, active,
11             created_on from users";
12         PreparedStatement preparedStatement = getPreparedStatement(qry);
13         List<User> users = executeQuery(preparedStatement);
14         return users;
15     }
16
17     public User findUserByEmail(String userEmail) throws Exception {
18         String qry = "select * from users where email = ?";
19         PreparedStatement preparedStatement = getPreparedStatement(qry);
20         preparedStatement.setString(1, userEmail);
21         return findUniqueResult(preparedStatement);
22     }
23
24     public User findUserByUsername(String username) throws Exception {
25         String qry = "select * from users where username =?";
26         PreparedStatement preparedStatement = getPreparedStatement(qry);
27         preparedStatement.setString(1, username);
28         return findUniqueResult(preparedStatement);
```

```
28     }
29
30     public User createUser(String username, String name, String email) throws
31         Exception {
32         String qry = "INSERT INTO users (username, name, email) VALUES (?, ?, ?)";
33         PreparedStatement preparedStatement = getPreparedStatement(qry);
34         preparedStatement.setString(1, username);
35         preparedStatement.setString(2, name);
36         preparedStatement.setString(3, email);
37         return createOrUpdateUser(username, preparedStatement);
38     }
39
40     public User updateUserEmail(User user, String newEmail) throws Exception {
41         String qry = "UPDATE users SET email = ? WHERE user_id = ? ";
42         PreparedStatement preparedStatement = getPreparedStatement(qry);
43         preparedStatement.setString(1, newEmail);
44         preparedStatement.setInt(2, user.getUserId());
45         return createOrUpdateUser(user.getUsername(), preparedStatement);
46     }
47
48     private User createOrUpdateUser(String username, PreparedStatement
49         preparedStatement) throws Exception {
50         int result = executeUpdateQuery(preparedStatement);
51
52         return findUserByUsername(username);
53     }
54
55     public void deleteUser(User user) throws Exception {
56         String qry = "DELETE FROM users WHERE user_id = ?";
57         PreparedStatement preparedStatement = getPreparedStatement(qry);
58         preparedStatement.setInt(1, user.getUserId());
59         createOrUpdateUser(user.getUsername(), preparedStatement);
60     }
61
62     private User findUniqueResult(PreparedStatement preparedStatement) throws
63         Exception {
64         List<User> users = executeQuery(preparedStatement);
65         if (users.isEmpty()) {
66             return null;
67         }
68         if (users.size() > 1) {
69             throw new Exception("Only one result expected");
70         }
71         return users.get(0);
72     }
73
74     private List<User> executeQuery(PreparedStatement preparedStatement) {
75         List<User> users = new ArrayList<>();
76
77         try (
78             ResultSet rs = preparedStatement.executeQuery()) {
79             while (rs.next()) {
80                 User user = buildUserFromResultSet(rs);
81                 users.add(user);
82             }
83         } catch (SQLException e) {
84             e.printStackTrace();
85         }
86         return users;
87     }
88
89     private PreparedStatement getPreparedStatement(String query) throws
90         SQLException {
91         if (getConnection() == null) {
92             try {
93                 setConnection(dBConnection.getConnection());
94             } catch (SQLException | IOException e) {
95                 e.printStackTrace();
96             }
97         }
98     }
```

```

93         }
94     }
95     return getConnection().prepareStatement(query);
96 }
97
98
99     private int executeUpdateQuery(PreparedStatement preparedStatement) {
100         int result = 0;
101         if (getConnection() == null) {
102             try {
103                 setConnection(dBConnection.getConnection());
104             } catch (SQLException | IOException e) {
105                 e.printStackTrace();
106             }
107         }
108         try {
109             result = preparedStatement.executeUpdate();
110         } catch (SQLException e) {
111             e.printStackTrace();
112         }
113         return result;
114     }
115
116     private User buildUserFromResultSet(ResultSet rs) throws SQLException {
117         int userId = rs.getInt("user_id");
118         String username = rs.getString("username");
119         String name = rs.getString("name");
120         String email = rs.getString("email");
121         int rank = rs.getInt("rank");
122         boolean active = rs.getBoolean("active");
123         Timestamp timestamp = rs.getTimestamp("created_on");
124         User user = new User(userId, username, name, email, rank, timestamp,
125                               active);
126         return user;
127     }
128
129     public Connection getConnection() {
130         return connection;
131     }
132
133     public void setConnection(Connection connection) {
134         this.connection = connection;
135     }

```

A l'apartat d'annexos de la unitat podeu trobar el codi després de la refactorització.

1.8 Què s'ha après?

Heu après que JDBC és la tecnologia que permet a una aplicació Java connectar-se a diferents bases de dades utilitzant una única interfície de programació.

Resumint, heu après a:

- Conèixer la utilitat i l'arquitectura de JDBC
- Usar una base de dades en memòria per fer tests unitaris
- Escriure el codi Java per poder fer operacions CRUD amb la base de dades

Ja esteu preparats per començar les activitats proposades en aquest apartat per tal de poder endinsar-vos en el món de la programació amb Java i bases de dades.

2. Accés a dades amb Java Enterprise Edition

Quan programeu en Java, la forma més simple per accedir a una base és mitjançant JDBC. Per desgràcia, amb JDBC es necessita una gran quantitat de treball manual per convertir els resultats d'una consulta a la base de dades en classes Java.

En el següent fragment de codi es mostra com transformar un resultat de consulta JDBC en un objecte User:

```

1 ResultSet rs = stmt.executeQuery(qry); {
2     while (rs.next()) {
3         int userId = rs.getInt("user_id");
4         String username = rs.getString("username");
5         String name = rs.getString("name");
6         String email = rs.getString("email");
7         int rank = rs.getInt("rank");
8         boolean active = rs.getBoolean("active");
9         Timestamp timestamp = rs.getTimestamp("created_on");
10        User user = new User(userId, username, name, email, rank, timestamp,
11                               active);
12        users.add(user);
13    }

```

Com podeu veure, hi ha molt codi repetitiu per obtenir els resultats dels diferents camps de la base de dades i transformar-los en variables de Java. Penseu que si aquesta classe tingués 30 atributs la quantitat de codi necessària augmentaria considerablement. Aquesta situació seria encara pitjor si a més a més de tenir 30 atributs estigués relacionada amb una altra classe. Per exemple, a l'aplicació “SocLoc” que construireu, un alumne (*User*) pot crear preguntes. Si la classe que representa les preguntes té 10 atributs, cada cop que fem una consulta on es retornin alumnes com a resultat hauríeu de repetir el codi anterior 40 vegades.

Un altre desavantatge de JDBC és la seva portabilitat. La sintaxi de la consulta canviàrà d'una base de dades a una altra. Per exemple, amb la base de dades Oracle la instrucció ROWNUM s'utilitza per limitar la quantitat de resultats retornats, mentre que SqlServer és TOP. El fet d'haver d'utilitzar SQL natiu específic per a cada BD fa que el nostre codi estigui acoblat amb el tipus de BD utilitzat. Això és un problema, ja que si en el futur es vol canviar de base de dades tindrem problemes. Hi ha diverses solucions a aquest tipus de problema, però totes passen per mantenir el codi SQL necessari per a cada tipus de BD; això costarà molt de mantenir i és molt propici a errors.

JPA (Java Persistence API) es va crear com una solució als problemes esmentats anteriorment. JPA permet treballar amb les classes de Java, ja que proporciona una capa transparent que s'encarrega de gestionar els detalls específics per a cada BD i permet focalitzar els esforços de desenvolupament en el codi Java. JPA representa una sèrie d'interfícies Java, així com una sèrie d'estàndards i especificacions que defineixen com han de ser les implementacions. JPA per si sol no farà res, necessitarà d'una implementació per poder ser utilitzada. Hi ha

moltes implementacions de JPA disponibles, tant gratuïtes com de pagament; per exemple, Hibernate, OpenJPA i EclipseLink.

La principal característica de JPA és la capacitat d'establir i gestionar les relacions entre les taules de la BD i les classes del codi Java. En Java, l'aplicació es modela través d'objectes, però les bases de dades relacionals només poden emmagatzemar valors escalars, com cadenes de caràcters o enters, i organitzar-los en taules. Sense JPA, tal com hem vist a l'exemple de JDBC, és el programador qui ha de convertir els valors representats en forma d'objectes en valors simples o agrupats per poder-los emmagatzemar a la BD, així com implementar el procés invers per poder extreure les dades. JPA defineix com s'ha de resoldre aquest problema i defineix com s'ha de fer el mapatge d'objectes relacionals (ORM Object-Relational Mapping).

Posarem les bases de l'aprenentatge configurant l'aplicació Java “SocLoc”. Explicarem:

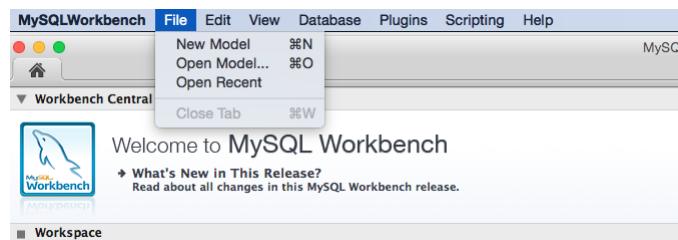
- Creació de la base de dades “SocLoc” a partir d'un model de MySQL Workbench
- Notacions JPA
- Tipus de dades a la BD i correspondència amb Java
- *Persistence units*
- Entorn Glassfish configurant el *datasource* i *connection pool*
- Validació
- Tests unitaris
- Utilitzar la base de dades en memòria H2 per escriure tests unitaris

2.1 "SocLoc". Dialogant amb clients amb JPA

El primer que haureu de fer serà importar la base de dades “SocLoc”, tal com es mostra en la figura 2.1.

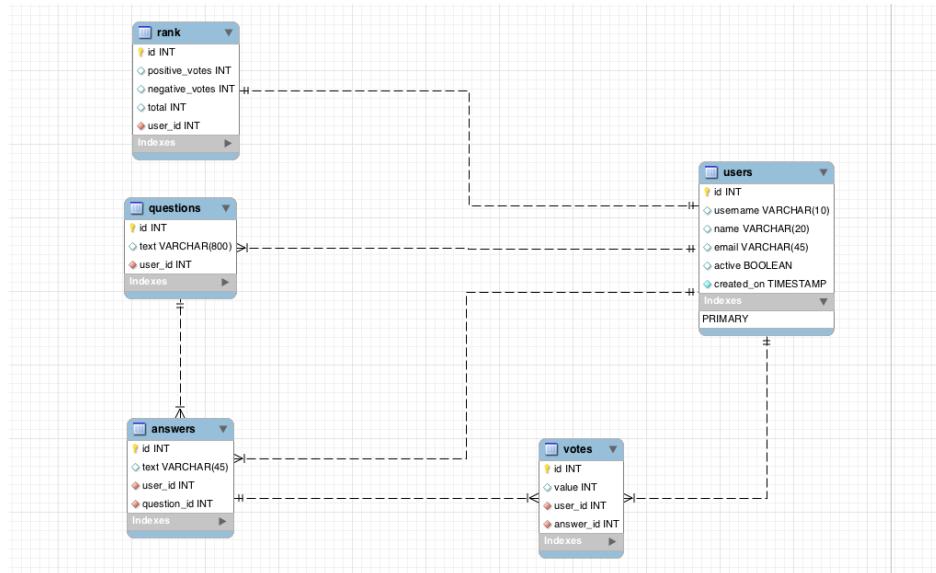
Podeu trobar la base de dades “SocLoc” al fitxer de MySQL Workbench que teniu disponible als annexos de la unitat.

FIGURA 2.1. Importar model de MySQL Workbench



Un cop importat, s'haurà d'haver importat el model entitat relació de la figura 2.2. El model ER de la base de dades “SocLoc” es compon de les següents taules:

- **rank**: serveix per guardar la puntuació que aconsegueix un alumne en base a les votacions que reben les respostes que dóna.
- **questions**: conté les preguntes dels alumnes.
- **answers**: guarda les respostes.
- **votes**: els alumnes que llegeixen una resposta la podran votar de forma positiva o negativa.
- **users**: guarda informació dels alumnes.

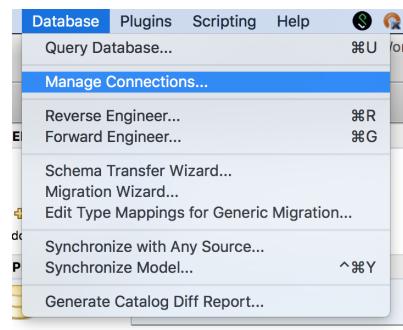
FIGURA 2.2. Model ER “Socloc”

A continuació haureu de connectar-vos al vostre servidor MySQL per tal de crear les taules que hem definit al model ER “SocLoc”. El primer pas és crear la base de dades al servidor MySQL. Hi ha diferents formes de fer-ho, però en aquest exemple utilitzarem el client de la línia de comandes. En el cas del servidor que s'està fent servir, aquestes són les dades de connexió:

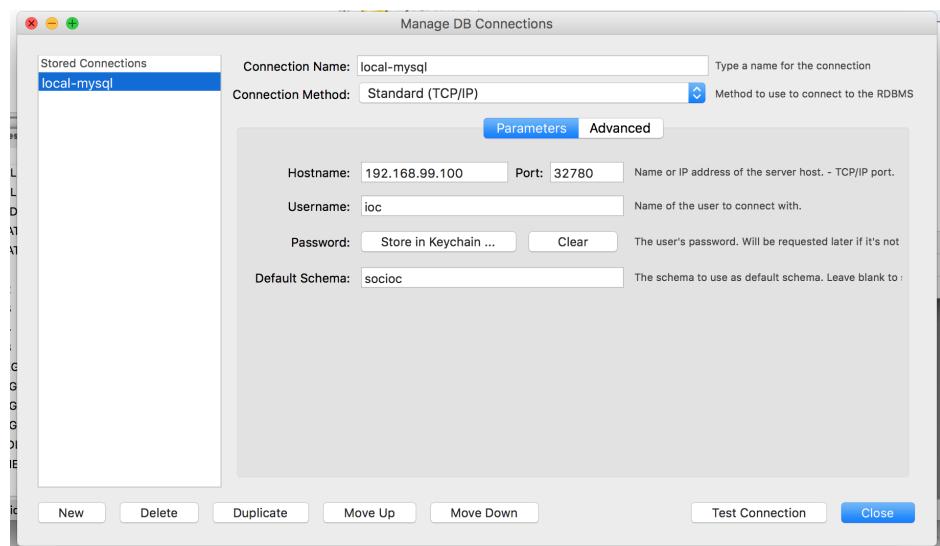
- usuari: root
- contrasenya: root
- IP del servidor: 192.168.99.100
- port: 32769

¹ `mysql --user=root --password=root --host=192.168.99.100 --port=32769`

Utilitzant Mysql Workbench, creeu una connexió amb el vostre servidor MySQL. Per fer-ho, seleccioneu *Manage connections*, tal com podeu veure en la figura 2.3.

FIGURA 2.3. Manage connections

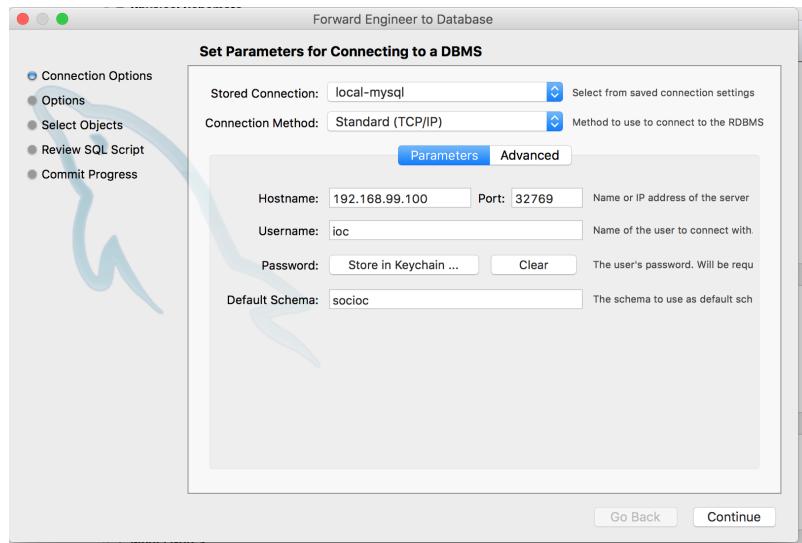
A continuació afegiu les dades de connexió. En la figura 2.4 podeu veure les dades de connexió al servidor local de MySQL.

FIGURA 2.4. Crear connexió amb DB

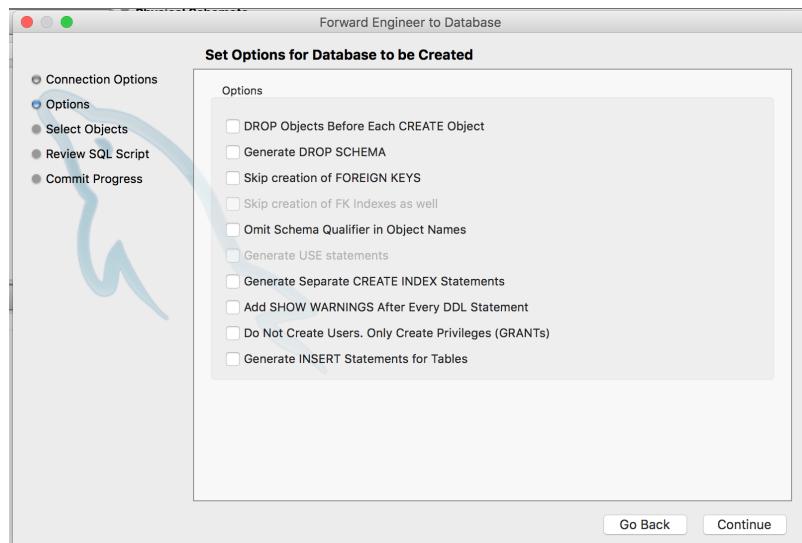
Un cop està configurada la connexió ja es pot passar a exportar el model al servidor. Seleccioneu l'opció *Forward Engineering* del menú *Database* (vegeu la figura 2.5).

FIGURA 2.5. Exportar taules al servidor

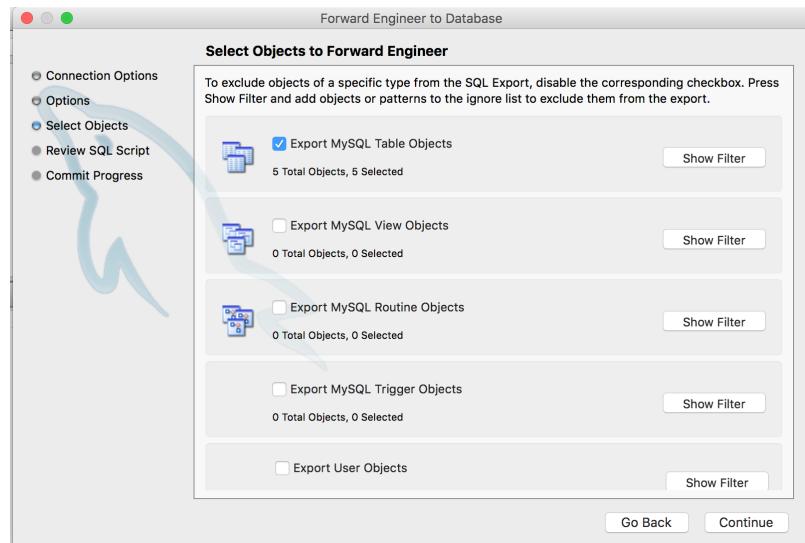
Seleccioneu la connexió que heu creat (vegeu la figura 2.6); en fer clic a *Continue* us demanarà la contrasenya per accedir a la BD.

FIGURA 2.6. Seleccioneu la connexió amb la BD

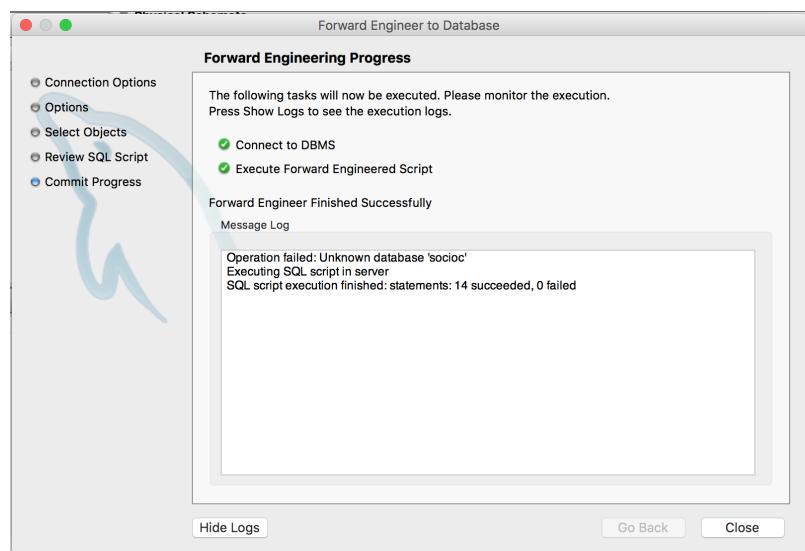
Quant a les opcions, podeu deixar les que hi ha per defecte (vegeu la figura 2.7).

FIGURA 2.7. Opcions d'exportació

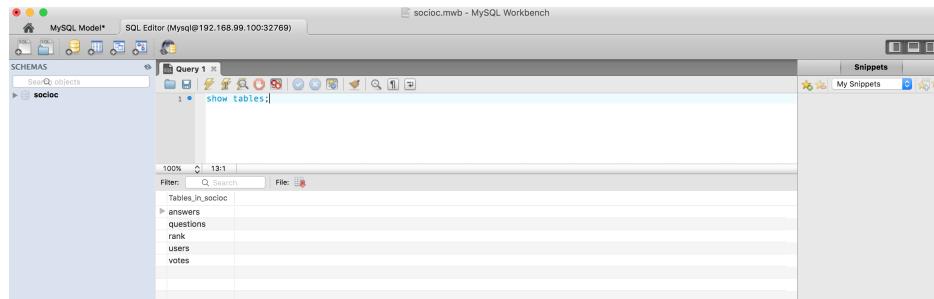
Com que només heu creat taules (vegeu la figura 2.8), només fa falta seleccionar la primera de les opcions, que exportarà les taules.

FIGURA 2.8. Opcions d'exportació

Els dos últims passos són una revisió dels *scripts* de creació dels objectes seleccionats (en el nostre cas, les taules). Si el procés de creació és correcte haureu de veure un missatge de confirmació com el de la figura 2.9.

FIGURA 2.9. Confirmació de la correcta exportació de les taules

Per comprovar que les taules s'han creat adequadament podeu utilitzar l'editor SQL de Workbench. Seleccioneu del menú *Database* l'opció *Query database*; després d'escollar la connexió amb la vostra BD accedireu a l'editor (vegeu la figura 2.10) i podreu executar una consulta perquè es mostrin totes les taules de la BD.

FIGURA 2.10. Editor SQL

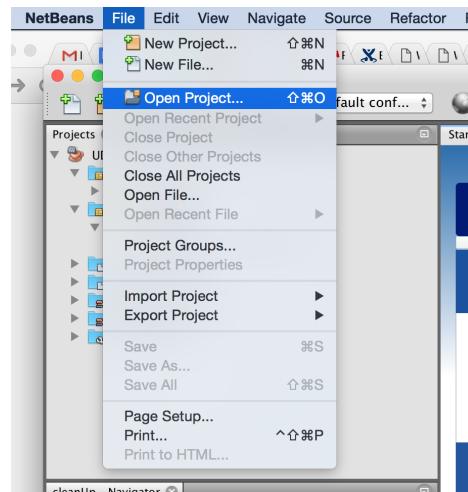
2.1.1 Anotacions JPA

Recordeu que JPA defineix com s'ha de fer el mapatge d'objectes relacionals (ORM Object-Relational Mapping) amb la base de dades. D'aquesta forma, quan desenvolueu una aplicació, com a programadors us podeu centrar en el codi Java i deixar que JPA s'encarregui dels detalls específics de la base de dades. Començareu creant una classe que representi un usuari de la nostra aplicació i que es pugui utilitzar per emmagatzemar dades.

Per fer això creareu una classe i l'anotareu amb l'anotació `@Entity`. Això transformarà una classe Java simple (POJO, Plain Old Java Object) en una EJB (Enterprise Java Bean). L'ús d'EJB permet gaudir dels serveis prestats pels servidors Java Enterprise Edition (Java EE). Hi ha diferents serveis que proporcionen el seu ús, com *clustering*, transaccionalitat a través de JTA, seguretat i connexions amb base de dades. A més de l'anotació `@Entity` hi ha altres anotacions que permeten definir quins atributs de la classe seran persistents i a quina columna de la taula seran emmagatzemats.

Un cop descarregat i descomprimit el fitxer de partida, ja el podeu importar; el nom del projecte és “UDF4-02”. En la figura 2.11 es mostra on és l'opció per obrir un projecte existent.

El fitxer de partida per fer anotacions JPA el trobareu als annexos de la unitat.

FIGURA 2.11. Editor SQL

Netbeans, en detectar el fitxer pom.xml, reconeixerà que és un projecte Maven i començarà a descarregar les dependències.

En el fitxer pom.xml de Maven, les úniques dependències que us fan falta són la del *driver* de la BD H2, de JUnit, javax.persistence i javax.validation. javax.persistence conté les llibreries amb totes les classes que necessitareu per treballar amb JPA. Encara que amb les llibreries de javax.persistence no necessitaríeu res més, javax.validation us aporta una sèrie d'anotacions que seran molt útils a l'hora de validar les dades que emmagatzemareu a la base de dades.

```

1 <dependencies>
2   <dependency>
3     <groupId>com.h2database</groupId>
4     <artifactId>h2</artifactId>
5     <version>1.4.190</version>
6   </dependency>
7   <dependency>
8     <groupId>junit</groupId>
9     <artifactId>junit</artifactId>
10    <version>4.12</version>
11  </dependency>
12  <dependency>
13    <groupId>javax.persistence</groupId>
14    <artifactId>persistence-api</artifactId>
15    <version>1.0.2</version>
16  </dependency>
17  <dependency>
18    <groupId>javax.validation</groupId>
19    <artifactId>validation-api</artifactId>
20    <version>1.1.0.Final</version>
21  </dependency>
22 </dependencies>
```

Com a programadors d'aplicacions orientades a objectes, el que us interessa és treballar amb objectes que representin les dades, més que haver de treballar directament amb SQL. JPA permet això, utilitzant ORM (Object Relational Mapping) per emmagatzemar i recuperar dades de la base de dades a través de l'ús de classes de tipus entitat (*entity classes*). Mitjançant l'anotació @Entity es transforma una classe Java simple (POJO, Plain Old Java Object) en una EJB (Enterprise Java Bean). L'ús d'EJB permet gaudir dels serveis prestats pels servidors Java Enterprise Edition (Java EE). Hi ha diferents serveis que proporciona el seu ús, com *clustering*, transaccionalitat a través JTA, seguretat i connexions amb base de dades. A més de l'anotació @Entity hi ha altres anotacions que permeten definir quins atributs de la classe seran persistents i a quina columna de la taula seran emmagatzemats. En el següent codi podeu veure les diferents anotacions que farem servir.

```

1 @Entity
2   @Table(name = "users")
3   public class User implements Serializable{
4     private static final long serialVersionUID = 1L;
5     @Id
6     @NotNull
7     @Column(name = "user_id")
8     private Long userId;
9
10    @NotNull
11    @Size(max = 30)
12    @Column(name = "username")
```

```
13  private String username;
14
15  @NotNull
16  @Size(max = 30)
17  @Column(name = "name")
18  private String name;
19
20  @NotNull
21  @Size(max = 30)
22  @Column(name = "email")
23  private String email;
24
25  @NotNull
26  @Size(max = 30, min = 5)
27  @Column(name = "password")
28  private String password;
29
30  @NotNull
31  @Column(name = "rank")
32  private Integer rank;
33
34  @NotNull
35  @Column(name = "active")
36  private Boolean active;
37
38  @NotNull
39  @Column(name = "created_on")
40  private Timestamp createdOn;
41
42  public User(){}
43
44  public Long getUserId() {
45      return userId;
46  }
47
48  public void setUserId(Long userId) {
49      this.userId = userId;
50  }
51
52  public String getUsername() {
53      return username;
54  }
55
56  public void setUsername(String username) {
57      this.username = username;
58  }
59
60  public String getName() {
61      return name;
62  }
63
64  public void setName(String name) {
65      this.name = name;
66  }
67
68  public String getEmail() {
69      return email;
70  }
71
72  public void setEmail(String email) {
73      this.email = email;
74  }
75
76  public String getPassword() {
77      return password;
78  }
79
80  public void setPassword(String password) {
81      this.password = password;
82  }
```

```

83
84     public Integer getRank() {
85         return rank;
86     }
87
88     public void setRank(Integer rank) {
89         this.rank = rank;
90     }
91
92     public Boolean getActive() {
93         return active;
94     }
95
96     public void setActive(Boolean active) {
97         this.active = active;
98     }
99
100    public Timestamp getCreatedOn() {
101        return createdOn;
102    }
103
104    public void setCreatedOn(Timestamp createdOn) {
105        this.createdOn = createdOn;
106    }
107}

```

L'anotació `@Entity` indica que serà una classe de tipus entitat i l'anotació `@Table` permet indicar quin és el nom de la taula que estarà lligada a aquesta classe. Un altre aspecte que és important destacar és que és sempre una bona idea fer que una classe de tipus entitat sigui serializable, per tal que la classe pugui ser passada per valor i no per referència. Això s'aconsegueix fent que la classe implementi la interfície `java.io.Serializable` i afegint l'atribut estàtic `serialVersionUID`. Sense entrar en molts detalls, `serialVersionUID` s'utilitza per comprovar que els objectes serialitzats i desserialitzats són compatibles.

Una classe del tipus entitat necessita tenir un constructor sense arguments i atributs per a cada una de les columnes de la taula. Les anotacions JPA que hem utilitzat les podeu veure en la taula 2.1.

TAULA 2.1. Annotacions JPA

Anotació	Descripció
<code>@Entity</code>	Transforma un POJO en una classe de tipus entitat per tal de poder utilitzar-la amb els serveis JPA.
<code>@Table</code> (opcional)	Especifica el nom de la taula de la base de dades associada amb l'entitat.
<code>@Id</code>	Designa un o més dels atributs de l'entitat com a la clau principal de la taula.
<code>@Column</code>	Associa un atribut que es vol emmagatzemar amb el nom d'una columna de la taula.

Hi ha altres anotacions que hem utilitzat i que no apareixen en la llista. Intenteu contestar a les següents preguntes abans de continuar:

- Quines són les anotacions que hem utilitzat i que no pertanyen a l'API de JPA?
- A quina API pertanyen?

- Per a què creieu que serveixen?

Les anotacions que no són part de JPA i que hem utilitzat (taula 2.2) són part de l'API javax.validation i serveixen per validar les dades que guardarem a la base de dades abans d'intentar guardar-les.

TAULA 2.2. Anotacions de validació

Anotació	Descripció
@Size	Permet especificar la longitud màxima i/o mínima d'una variable de tipus <i>string</i> .
@NotNull	Estableix que l'atribut not pot tenir un valor <i>null</i> .

Com ja hem vist, JPA és una especificació que indica com es transformen classes en entitats i com aquestes es poden fer servir per llegir i escriure en una base de dades. És una especificació però no una implementació, és a dir, defineix com han de ser les operacions però no les implementa. Això vol dir que amb l'exemple de la classe User que hem vist no podem fer cap operació amb la base de dades. Per poder fer-ho hi ha dues opcions: utilitzar Persistence Units (unitats de persistència) o emprar Enterprise Java Beans, que requerirà de l'ús d'un servidor d'aplicacions com pot ser Glassfish.

2.1.2 Unitats de persistència

Les unitats de persistència són les peces que enllacen l'entitat que hem definit amb la base de dades. Poden usar un *pool* de connexions amb la BD definides al servidor d'aplicacions o, com en l'exemple següent, una connexió JDBC. El que es aconseguir és crear una unitat de persistència que pugueu utilitzar en els tests unitaris. Obriu el codi que crearà el projecte UDF4-02. Al fitxer src/test/resources/META-INF/persistence.xml trobareu la configuració de la unitat de persistència.

Als annexos de la unitat trobareu l'arxiu amb el codi per crear el projecte UDF4-02.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5      http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
6      <persistence-unit name="InMemoryH2PersistenceUnit" transaction-type=
7          RESOURCE_LOCAL>
8          <class>org.ioc.daw.user.User</class>
9          <properties>
10             <property name="javax.persistence.jdbc.driver" value="org.h2.Driver
11                 "/>
12             <property name="javax.persistence.jdbc.user" value="username"/>
13             <property name="javax.persistence.jdbc.password" value="password"/>
14             <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:
15                 socioc_db"/>
16         </properties>
17     </persistence-unit>
18 </persistence>
```

El fitxer XML que defineix la unitat de persistència és on resideix informació que necessita JPA per saber com connectar-se a la BD. Aquest fitxer pot contenir la configuració de com connectar-se a múltiples BD. Cadascuna d'aquestes configuracions indicarà el tipus de transacció que s'utilitzarà a l'hora de fer les operacions amb la BD que serà JTA o RESOURCE_LOCAL.

A *persistence-unit name=* s'indica el nom de la unitat de persistència, que serà utilitzat a l'aplicació per obtenir una referència a la configuració que defineix com ens connectem a la BD. El tipus de transacció indica si s'utilitzarà Java Transaction API (JTA) *entity managers* (gestors d'entitats) per ser utilitzats en un servidor d'aplicacions o bé *entity managers* locals que no necessiten de cap servidor d'aplicacions, i que és el tipus que us interessa per als vostres tests unitaris.

<class> permet definir quines classes (en el vostre cas només n'hi ha una, de moment) seran mapejades amb la BD. Després es defineixen els elements que permeten la connexió amb la BD. En aquest cas, nom d'usuari i contrasenya per accedir a la BD i l'*string* de connexió que defineix com accedir a la base de dades.

Un cop heu establert com es farà la connexió amb la base de dades, ja podeu escriure un test unitari.

```

1  public class UserTest {
2      private EntityManager entityManager;
3      private EntityTransaction entityTransaction;
4
5      @Before
6      public void setUp() {
7          entityManager = Persistence.createEntityManagerFactory(
8              "InMemoryH2PersistenceUnit").createEntityManager();
9          entityTransaction = entityManager.getTransaction();
10     }
11
12     @After
13     public void cleanUp() {
14         entityManager.close();
15     }
16
17     @Test
18     public void findAllUsers(){
19
20         User user = new User();
21         user.setActive(true);
22         user.setCreatedOn(new Timestamp(new Date().getTime()));
23         user.setEmail("test@test.com");
24         user.setName("Jane");
25         user.setPassword("password");
26         user.setRank(100);
27         user.setUsername("jdoe");
28         user.setUserId(23L);
29
30         entityTransaction.begin();
31         entityManager.persist(user);
32         entityTransaction.commit();
33
34         Query query = entityManager.createNativeQuery("select * from users",
35             User.class);
36         List<User> userList = query.getResultList();
37         Assert.assertEquals(1, userList.size());
38         Assert.assertEquals("jdoe", userList.get(0).getUsername());
39     }
40

```

39 }

Amb `Persistence.createEntityManagerFactory` definiu quina serà la unitat de persistència que necessitareu per crear l'`EntityManager`. Després creeu un objecte per fer transaccions amb la BD; és part del mètode `setUp`, perquè aquest objecte l'utilitzareu en tots els tests. Al test `findAllUsers` primer creeu un objecte `User` (que, recordeu, serà una entitat), inicieu la transacció amb la BD, salveu l'objecte amb `entityManager.persist(user)`; i finalment dieu a l'objecte que gestiona les transaccions que apliqui tots els canvis a la BD. Fixeu-vos que no serà fins a `entityTransaction.commit()` que els canvis s'escriuran a la base de dades. Finalment, el test comprova que a la taula “Users” de la BD només hi ha una entrada i que correspon a l’usuari que acabeu de crear. Executeu el test i obtindreu el següent error:

```
1 javax.persistence.PersistenceException: No resource files named META-INF/
  services/javax.persistence.spi.PersistenceProvider were found. Please make
  sure that the persistence provider jar file is in your classpath.
2
3   at javax.persistence.Persistence.findAllProviders(Persistence.java:167)
```

Quin és el motiu pel qual teniu aquest error?

Com ja heu vist, JPA només és una especificació, però no una implementació. L’error vol dir exactament això: quan s’intenta executar el codi no es troba cap llibreria (`PersistenceProvider`) que implementi JPA, per la qual cosa resulta impossible fer cap operació amb la base de dades. Ho solucionareu utilitzant la implementació [EclipseLink](#). No entrarem en detalls sobre EclipseLink, ja que més endavant utilitzareu Hibernate com a implementació.

Per incloure EclipseLink al nostre projecte, el primer serà modificar el fitxer de dependències `pom.xml`.

```
1 <dependency>
2   <groupId>org.eclipse.persistence</groupId>
3   <artifactId>eclipselink</artifactId>
4   <version>2.5.0</version>
5 </dependency>
```

A continuació haureu de modificar la unitat de persistència per indicar quina implementació utilitzareu:

```
1 <persistence-unit name="InMemoryH2PersistenceUnit" transaction-type="
  RESOURCE_LOCAL">
2   <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
3   <class>org.ioc.daw.user.User</class>
4   <properties>
5     <property name="javax.persistence.jdbc.driver" value="org.h2.Driver
      "/>
6     <property name="javax.persistence.jdbc.user" value="username"/>
7     <property name="javax.persistence.jdbc.password" value="password"/>
8     <property name="javax.persistence.jdbc.url" value="
      jdbc:h2:mem:socioc_db;INIT=runscript from 'classpath:init.sql'
      "/>
9     <property name="javax.persistence.schema-generation.database.action
      " value="drop-and-create"/>
10    <property name="javax.persistence.schema-generation.create-source"
      value="metadata"/>
```

```

11         <property name="javax.persistence.schema-generation.drop-source"
12             value="metadata"/>
13     </properties>
  </persistence-unit>

```

A la línia 2 s'ha afegit quina serà la implementació del PersistenceProvider. A la línia 8, on s'indica l'*string* de connexió, afegim també que s'executi un *script* on creareu la taula “Users”, que utilitzareu per emmagatzemar els usuaris del nostre sistema. El fitxer src/test/resources/init.sql té el següent contingut:

```

1 CREATE TABLE users(user_id INT PRIMARY KEY AUTO_INCREMENT NOT NULL,
2                     username VARCHAR(30) NOT NULL,
3                     name VARCHAR(20) NOT NULL,
4                     email VARCHAR(50) NOT NULL,
5                     password VARCHAR(50) NOT NULL,
6                     rank INT DEFAULT 0,
7                     active BOOLEAN DEFAULT true,
8                     created_on TIMESTAMP AS CURRENT_TIME)

```

Torneu a executar el test findAllUsers, que aquest cop hauria de passar. Un problema d'aquest codi és que la lògica per emmagatzemar i recuperar usuaris està al test. És important que aquest codi estigui a la seva pròpia classe per tal que es pugui utilitzar en tota l'aplicació. Creareu una classe, UserService, que s'encarregarà d'aquesta funció i que tindrà la lògica necessària per guardar un usuari a la base de dades, modificar-lo i buscar usuaris pel seu nom. Començareu creant una interfície que definirà les operacions relacionades amb la BD. Programar utilitzant interfícies té avantatges, com heu pogut veure amb JPA. Es defineix una interfície i després es pot canviar la implementació; així, si canviéssiu de BD i per exemple escollíssiu una BD NoSQL on no es pugui utilitzar JPA només hauríeu de preocupar-vos de canviar la implementació. Definiu la interfície org.ioc.daw.user.UserDAO que us permeti guardar usuaris a la BD, esborrar-los, modificar-los i trobar un usuari pel seu nom.

```

1 public interface UserService {
2     public void create(User user);
3     public void edit(User user);
4     public void remove(User user);
5     public User findUserByUsername(String username);
6 }

```

A continuació creeu la seva implementació:

```

1 public class UserServiceImpl implements UserService {
2     private EntityManager entityManager;
3
4     public UserServiceImpl(EntityManager entityManager) {
5         this.entityManager = entityManager;
6     }
7
8     @Override
9     public void create(User user) {
10         entityManager.persist(user);
11     }
12
13     @Override
14     public void edit(User user) {
15         entityManager.merge(user);
16     }
17

```

```

18     @Override
19     public void remove(User user) {
20         entityManager.remove(user);
21     }
22
23     @Override
24     public User findUserByUsername(String username) {
25         return (User) entityManager.createQuery("select object(o) from User o " +
26             "where o.username = :username")
27             .setParameter("username", username)
28             .getSingleResult();
29     }
30 }
```

Els mètodes `create`, `remove` i `edit` defineixen les operacions JPA per guardar, actualitzar i esborrar entitats de la BD. La part més interessant és la del mètode `findUserByUsername`. Utilitzeu `createQuery`, i en aquest cas feu servir una consulta parametrizada on indiqueu la classe de l'objecte que retornarà la consulta, el paràmetre que passareu (`username`) i que només retornarà un resultat, ja que només podeu tenir un usuari amb un nom determinat. El test el podeu implementar de la següent manera (en aquest cas emmagatzemeu dos usuaris):

```

1  public class UserServiceTest {
2      private EntityManager entityManager;
3      private EntityTransaction entityTransaction;
4      private UserService userService;
5
6      @Before
7      public void setUp() {
8          entityManager = Persistence.createEntityManagerFactory(
9              "InMemoryH2PersistenceUnit").createEntityManager();
10         userService = new UserServiceImpl(entityManager);
11         entityTransaction = entityManager.getTransaction();
12     }
13
14     @After
15     public void cleanUp() {
16         entityManager.close();
17     }
18
19     @Test
20     public void findAllUsers(){
21         String username = "jdoe";
22         User user = new User();
23         user.setActive(true);
24         user.setCreatedOn(new Timestamp(new Date().getTime()));
25         user.setEmail("test@test.com");
26         user.setName("Jane");
27         user.setPassword("password");
28         user.setRank(100);
29         user.setUsername(username);
30         User user1 = new User();
31         user1.setActive(true);
32         user1.setCreatedOn(new Timestamp(new Date().getTime()));
33         user1.setEmail("test1@test.com");
34         user1.setName("Joe");
35         user1.setPassword("password");
36         user1.setRank(100);
37         user1.setUsername("joeTest");
38
39         entityTransaction.begin();
40         userService.create(user);
41         userService.create(user1);
42         entityTransaction.commit();
43
44         User userFromDB = userService.findUserByUsername(username);
```

```

44     Assert.assertNotNull(userFromDB);
45     Assert.assertEquals("jdoe", userFromDB.getUsername());
46     Assert.assertEquals("test@test.com", userFromDB.getEmail());
47     Assert.assertNotNull(userFromDB.getUserId());
48 }
49 }
```

Fixeu-vos que en aquest test no heu donat valor a `userId`, però tal com testeja `Assert.assertNotNull(userFromDB.getUserId());` sí que té un valor. Quina anotació JPA s'ha d'afegir a l'atribut `userId` de la classe `User` que generarà automàticament un valor? La resposta és `GeneratedValue`.

```

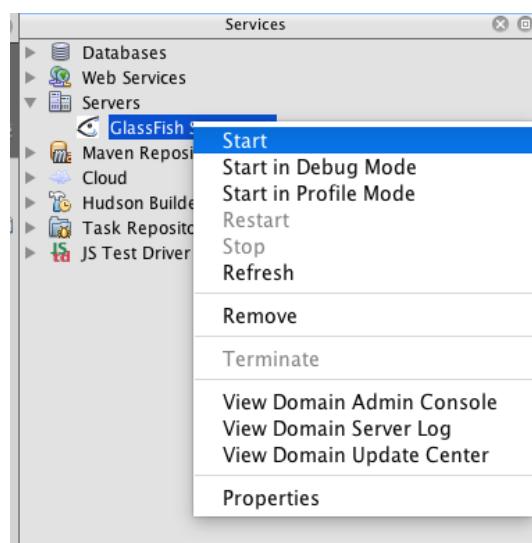
1 @Id
2   @NotNull
3   @GeneratedValue
4   @Column(name = "user_id")
5   private Long userId;
```

2.2 Servidor d'aplicacions Glassfish

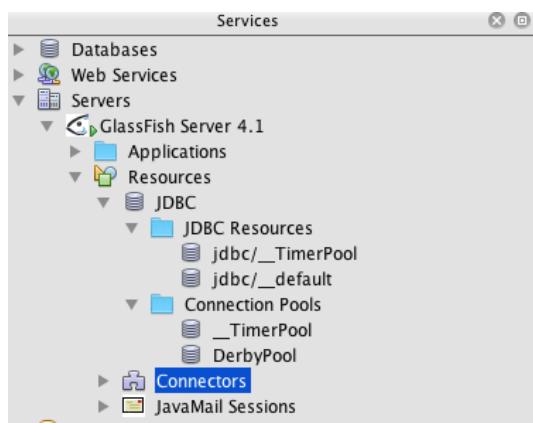
Quan vulgueu desplegar la vostra aplicació, utilitzar EclipseJPA com a implementació de JPA no serà suficient. Necessiteu un servidor d'aplicacions que permeti desplegar aplicacions Java EE. Glassfish és el servidor d'aplicacions de referència per a aplicacions Java EE i inclou les tecnologies EJB, JPA, JSF (Java Server Faces) i JMS (Java Messaging System). Netbeans inclou per defecte un servidor Glassfish. A la pestanya *Serveis*, tal com podeu veure en la figura 2.11, ja hi ha un servidor Glassfish configurat.

En algunes versions del servidor Glassfish hi ha un bug no solucionat en afegir un pool de connexions. Utilitzeu la versió 4.0 o superior, que podeu descarregar de: glassfish.java.net/download-archive.html.

FIGURA 2.12. Glassfish

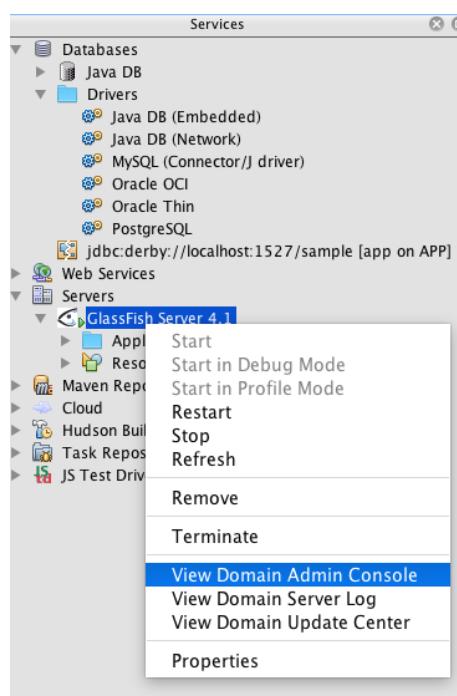


Arranqueu el servidor; després d'uns segons estarà funcionant, i si feu clic a *Resources* veureu que hi ha dues carpetes, *JDBC Resources* i *Connection Pools* (vegeu la figura 2.13).

FIGURA 2.13. 'Resources and connection pools'

2.2.1 'Pool' de connexions

Un *pool* de connexions és un sèrie de connexions amb la BD que es guarden en *cache* i que poden ser reutilitzades per diferents consultes que es facin a la BD. El motiu d'utilitzar-les és que milloren notablement el rendiment de l'aplicació i la fan molt més ràpida. La gestió d'obrir i tancar connexions amb al BD cada cop que es necessiti guardar o recuperar dades és molt costosa, especialment en aplicacions dinàmiques on el contingut es genera dinàmicament a partir de les dades de la BD. La forma de funcionar és que quan es crea una connexió es posa al *pool*, de manera que quan es torni a necessitar la connexió no s'ha de tornar a establir. El que volem fer és establir un *pool* de connexions amb la base de dades MySQL. Per fer-ho, un cop el servidor Glassfish ha arrencat, accediu a la consola d'administració (vegeu la figura 2.14).

FIGURA 2.14. 'Resources and connection pools'

La consola d'administració és una aplicació web, així que s'obrirà una finestra al vostre navegador. Navegueu a Resources/JDBC/JDBC Connection Pools i veureu els *pools* de connexions creats per defecte (vegeu la figura 2.15).

FIGURA 2.15. 'Resources and connection pools'

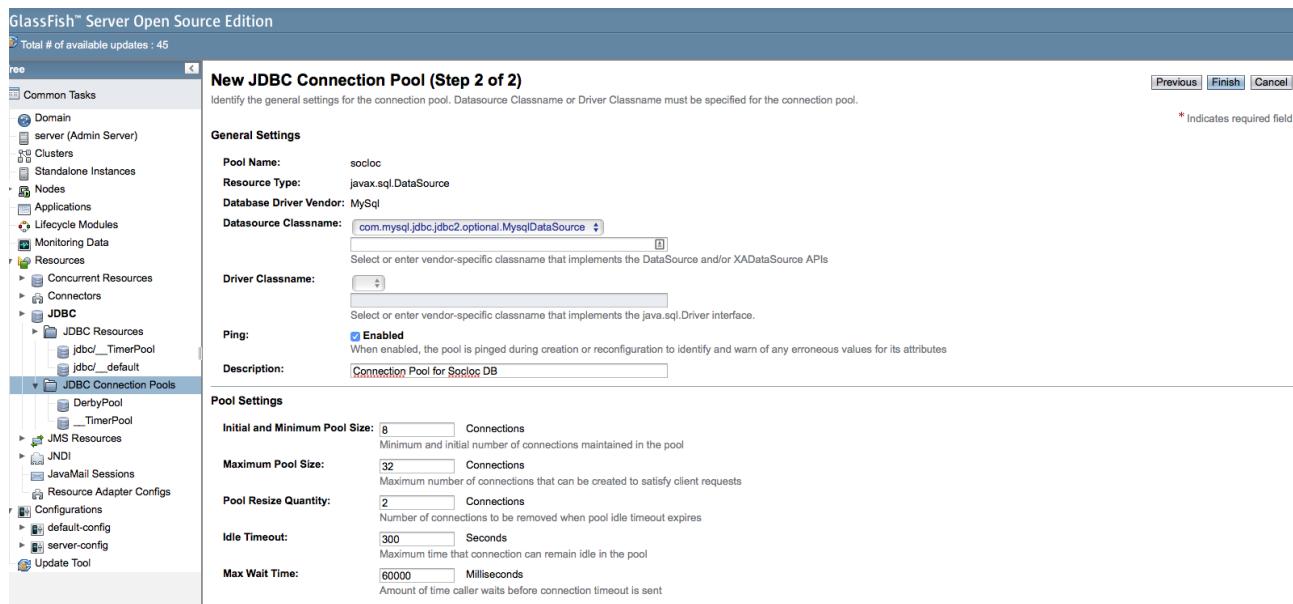
Select	Pool Name	Resource Type	Classname	Description
<input type="checkbox"/>	DerbyPool	javax.sql.DataSource	org.apache.derby.jdbc.ClientDataSource	
<input type="checkbox"/>	TimerPool	javax.sql.XADataSource	org.apache.derby.jdbc.EmbeddedXADataSource	

Feu clic a *New* i podreu crear el *pool* de connexions. Es fa en dues parts. En la primera part es dóna nom al *pool* de connexions i se selecciona el tipus de recurs i el *driver* que s'utilitzarà per connectar-se amb la base de dades (vegeu la figura 2.16).

FIGURA 2.16. Configuració del 'pool' de connexions I

A continuació es configuren els paràmetres del *pool* de connexions (vegeu la figura 2.17). Els que hi ha per defecte ja aniran bé, així que no els canviareu. Cal destacar els següents:

- Initial and Minimum Pool Size: nombre mínim de connexions amb la BD que mantindrem al *pool*.
- Maximum Pool Size: nombre màxim de connexions simultànies que mantindrem al *pool*.
- Pool Resize Quantity: nombre de connexions que es trauran del *pool* quan no hi hagi activitat durant més del temps establert pel paràmetre Idle Timeout.
- Idle Timeout: temps màxim que una connexió pot estar inactiva.

FIGURA 2.17. Configuració del 'pool' de connexions II

Els únics valors que heu de canviar són els relatius al nom de la base de dades, l'usuari, la contrasenya, la IP i el port que utilitza el servidor MySQL, que trobareu al final de la pàgina web (vegeu la figura 2.18).

FIGURA 2.18. Configuració del 'pool' de connexions III

Additional Properties (5)			
	Add Property	Delete Properties	
Select	Name	Value	Description
<input type="checkbox"/>	password	ioc	
<input type="checkbox"/>	databaseName	socioc	
<input type="checkbox"/>	serverName	192.168.99.100	
<input type="checkbox"/>	user	ioc	
<input type="checkbox"/>	portNumber	32769	

Un cop configurat, assegureu-vos que la connexió amb la BD funciona utilitzant el botó *Ping*. Si obtenuï l'error mostrat en la figura 2.19 és perquè el servidor Glassfish no pot accedir al *driver* de connexió amb MySQL.

FIGURA 2.19. Connector MySQL no trobat

En aquest cas feu el següent:

1. Descarregueu-vos el connector de dev.mysql.com/downloads/connector/j/5.1.html.
2. Descomprimiu-lo i copieu l'arxiu mysql-connector-java-5.1.40-bin.jar a `/instalacio-del-servidor-glassfish/glassfish-4.1/glassfish/domains/domain1/lib/`. Glassfish estarà instal·lat en el directori on hagueu instal·lat Netbeans. Per veure on està instal·lat exactament mireu les propietats del servidor Glassfish (*Installation Location*) (vegeu la figura 2.20 i la figura 2.21).

FIGURA 2.20. Directori d'instal·lació de Glassfish

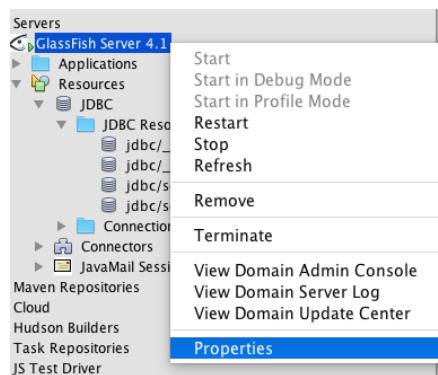
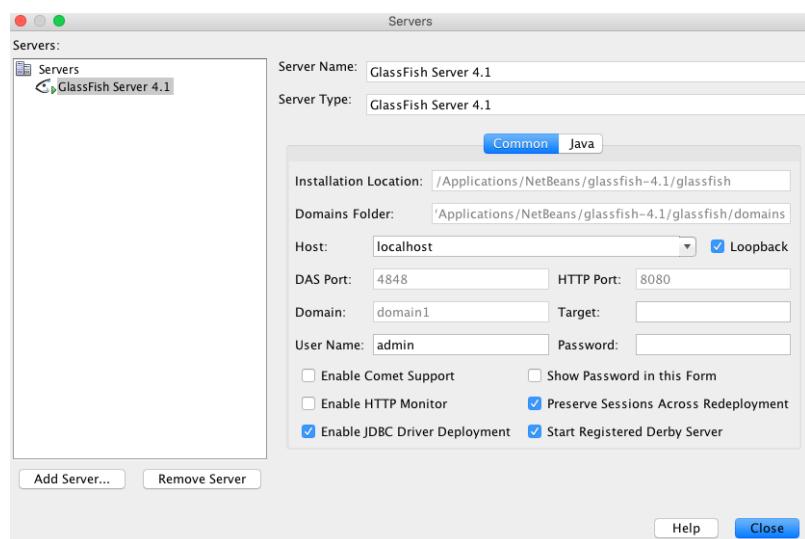
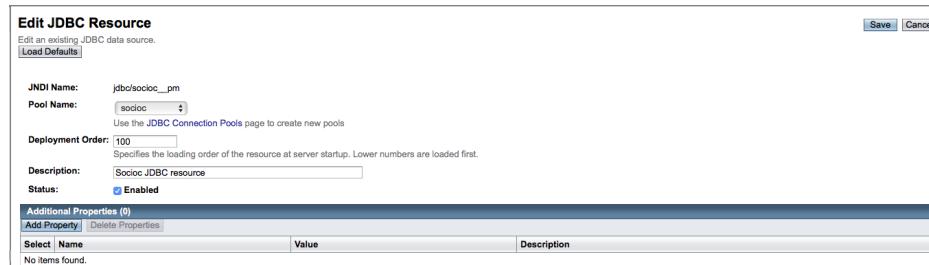


FIGURA 2.21. Directori d'instal·lació de Glassfish

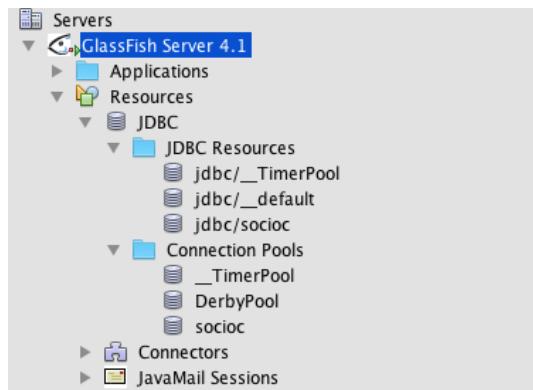


2.2.2 Recursos JDBC

Un recurs JDBC proporciona a una aplicació la forma de connectar-se a una base de dades. Típicament, l'administrador del servidor Glassfish creará un recurs JDBC que establirà quin dels *pools* de connexions utilitzarà l'aplicació. Cada recurs JDBC creat al servidor d'aplicacions tindrà un únic identificador JNDI (Java Naming and Directory Interface). JNDI és un servei ofert pels servidors d'aplicacions Java EE que permet als clients (en aquest cas, l'aplicació que estem desenvolupant) descobrir serveis i objectes utilitzant un nom. L'objecte que la vostra aplicació descobrirà serà l'objecte que representa el *pool* de connexions. A l'hora d'escollar el nom per al recurs JDBC es pot triar qualsevol, però per convenció s'utilitza `jdbc/nom_del_recurs_pm`. Per crear un recurs JDBC aneu a la consola d'administració de Glassfish i navegueu a *Resources/JDBC/JDBC Resources*. Amb el botó *New*, creeu un nou recurs i afegiu-hi les dades, tal com podeu veure en la figura 2.22.

FIGURA 2.22. Creació d'un recurs JDBC

Si refresqueu la informació del servidor Glassfish a Netbeans hauríeu de veure tant el recurs JDBC creat com el *pool* de connexions (vegeu la figura 2.23).

FIGURA 2.23. Recurs JDBC i 'pool' de connexions

2.2.3 Connectar l'aplicació "Socloc" amb el recurs JDBC

Un cop ja teniu el *pool* de connexions i el recurs JDBC creat, ja podeu fer que la vostra aplicació els utilitzi. Per fer-ho haureu de definir una unitat de persistència que especifiqui el recurs JDBC que utilitzareu. A partir del codi , importeu el projecte UDF4-02 i editeu el fitxer persistence.xml i amb el següent contingut:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5          http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
6      <persistence-unit name="InMemoryH2PersistenceUnit" transaction-type="
7          RESOURCE_LOCAL">
8          <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
9          <class>org.ioc.daw.user.User</class>
10         <properties>
11             <property name="javax.persistence.jdbc.driver" value="org.h2.Driver
12                 "/>
13             <property name="javax.persistence.jdbc.user" value="username"/>
14             <property name="javax.persistence.jdbc.password" value="password"/>
15             <property name="javax.persistence.jdbc.url" value="
16                 jdbc:h2:mem:socio_c_db;INIT=runscript from 'classpath:init.sql'
17                 "/>
18             <property name="javax.persistence.schema-generation.database.action
19                 " value="drop-and-create"/>
20             <property name="javax.persistence.schema-generation.create-source"
21                 value="metadata"/>
22             <property name="javax.persistence.schema-generation.drop-source"
23                 value="metadata"/>

```

Trobareu el codi per connectar l'aplicació "Socloc" amb el recurs JDBC als annexos de la unitat.

```

17      </properties>
18  </persistence-unit>
19  <persistence-unit name="MysqlResourceSocIocPersistence" transaction-type="
20    JTA">
21    <jta-data-source>jdbc/socioc_pm</jta-data-source>
22    <class>org.ioc.daw.user.User</class>
23    <properties>
24      <property name="eclipselink.logging.level" value="FINEST"/>
25    </properties>
26  </persistence-unit>
</persistence>

```

Hi ha diversos canvis. El primer és que la definició de la nova unitat de persistència `MysqlResourceSocIocPersistence` és molt més simple. Penseu que ara tots els detalls de la connexió estan gestionats per Glassfish, així que vosaltres només us heu de preocupar d'indicar quin serà el recurs JDBC que utilitzareu. L'única propietat que s'ha afegit a la línia 23 és `eclipselink.logging.level`, que indica el nivell de *logging* de l'aplicació i que serà útil per solucionar problemes. Al test unitari anterior, la gestió de les transaccions era part del test.

```

1 entityTransaction.begin();
2   userService.create(user);
3   userService.create(user1);
4   entityTransaction.commit();

```

Això no hauria de formar part del test, ja que el que us interessa és que sigui la classe que fa les consultes a la BD la que s'encarregui de gestionar les transaccions. A més a més, en utilitzar un servidor d'aplicacions les vostres classes seran ara entitats de ple dret, per la qual cosa es pot aprofitar per simplificar el codi utilitzant anotacions. Per fer-ho haureu d'afegir la següent dependència al fitxer pom.xml.

```

1 <dependency>
2   <groupId>javax.ejb</groupId>
3   <artifactId>javax.ejb-api</artifactId>
4   <version>3.2</version>
5 </dependency>

```

Vegeu com s'ha de modificar la classe `UserServiceImpl`:

```

1 @Stateless
2 @TransactionManagement(TransactionManagementType.BEAN)
3 public class UserServiceImpl implements UserService {
4     @PersistenceContext(unitName = "MysqlResourceSocIocPersistence")
5     private EntityManager entityManager;
6
7     @Resource
8     private EJBContext context;
9
10    @Override
11    public void create(User user) {
12        UserTransaction utx = context.getUserTransaction();
13        try {
14            utx.begin();
15            entityManager.persist(user);
16            utx.commit();
17        } catch (Exception e) {
18            e.printStackTrace();
19            try {
20                utx.rollback();
21            } catch (Exception e1) {
22                e1.printStackTrace();
23            }
24        }

```

```

25     }
26
27     @Override
28     public void edit(User user) {
29         UserTransaction utx = context.getUserTransaction();
30         try {
31             utx.begin();
32             entityManager.merge(user);
33             utx.commit();
34         } catch (Exception e) {
35             try {
36                 utx.rollback();
37             } catch (Exception e1) {
38                 e1.printStackTrace();
39             }
40             e.printStackTrace();
41         }
42     }
43
44     @Override
45     public void remove(User user) {
46         UserTransaction utx = context.getUserTransaction();
47         try {
48             utx.begin();
49             entityManager.remove(user);
50             utx.commit();
51         } catch (Exception e) {
52             try {
53                 utx.rollback();
54             } catch (Exception e1) {
55                 e1.printStackTrace();
56             }
57             e.printStackTrace();
58         }
59     }
60
61     @Override
62     public User findUserByUsername(String username) {
63         return (User) entityManager.createQuery("select object(o) from User o " +
64             "where o.username = :username")
65             .setParameter("username", username)
66             .getSingleResult();
67     }

```

L'anotació `Stateless` transforma un POJO en una EJB de sessió que adquiereix la capacitat d'executar operacions en un servidor d'aplicacions (Glassfish, en el vostre cas). `TransactionManagement` és necessària per permetre fer operacions amb la BD a través del servidor d'aplicacions. En aquest cas, farà que hi hagi disponible al context de EJB (`EJBContext`) la classe `UserTransaction`, que utilitzareu per indicar quan comencen i acaben les transaccions amb la BD. L'anotació `PersistenceContext` permet declarar quina serà la unitat de persistència que utilitzareu per indicar com l'EJB `UserService` s'ha de connectar amb la BD. Finalment, a cadascun dels mètodes que s'encarreguen de modificar dades a la BD utilitzeu `UserTransaction` per establir quan comencen i acaben les transaccions. Hi ha un mètode, `utx.rollback()`, que desfarà les dades modificades a la BD en cas que hi hagi un error.

A continuació cal modificar el vostre test unitari. Hi ha un problema: volem que el test utilitzi el recurs JDBC que heu creat al servidor Glassfish. Això vol dir que el vostre test ha de poder accedir a les EJB a Glassfish. Per aconseguir-ho utilitzareu un servidor Glassfish-embedded, que s'iniciarà durant els tests i permetrà accedir als recursos JDBC i EJB del servidor real Glassfish. El primer pas és afegir unes dependències al fitxer pom.xml. Fixeu-vos que heu de substituir

DIRECTORI_INSTALACIO_NETBEANS pel directori on tingueu instal·lat el servidor Glassfish (vegeu la figura 2.20 i la figura 2.21). També heu de copiar el fitxer mysql-connector-java-5.1.40-bin.jar al directori /NetBeans/glassfish-4.1/glassfish/lib/embedded.

```

1 <properties>
2   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3   <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
4   <glassfish.embedded-static-shell.jar>/DIRECTORI_INSTALACIO_NETBEANS/
5     glassfish-4.1/glassfish/lib/embedded/glassfish-embedded-static-shell.
6       jar</glassfish.embedded-static-shell.jar>
7   <glassfish.mysql.jar>/DIRECTORI_INSTALACIO_NETBEANS/glassfish-4.1//.
8     glassfish/lib/embedded/mysql-connector-java-5.1.40-bin.jar</glassfish.
9       mysql.jar>
10  </properties>
11
12  <dependencies>
13    <dependency>
14      <groupId>com.h2database</groupId>
15      <artifactId>h2</artifactId>
16      <version>1.4.190</version>
17    </dependency>
18    <dependency>
19      <groupId>junit</groupId>
20      <artifactId>junit</artifactId>
21      <version>4.12</version>
22    </dependency>
23    <dependency>
24      <groupId>javax.validation</groupId>
25      <artifactId>validation-api</artifactId>
26      <version>1.1.0.Final</version>
27    </dependency>
28    <dependency>
29      <groupId>org.eclipse.persistence</groupId>
30      <artifactId>eclipselink</artifactId>
31      <version>2.5.0</version>
32    </dependency>
33    <dependency>
34      <groupId>org.glassfish.main.extras</groupId>
35      <artifactId>glassfish-embedded-all</artifactId>
36      <version>4.1.1</version>
37      <scope>system</scope>
38      <systemPath>${glassfish.embedded-static-shell.jar}</systemPath>
39    </dependency>
40    <dependency>
41      <groupId>javax.ejb</groupId>
42      <artifactId>javax.ejb-api</artifactId>
43      <version>3.2</version>
44    </dependency>
45    <dependency>
46      <groupId>mysql</groupId>
47      <artifactId>mysql-connector-java</artifactId>
48      <version>5.1.40</version>
49      <scope>system</scope>
50      <systemPath>${glassfish.mysql.jar}</systemPath>
51    </dependency>
52  </dependencies>
```

El directori glassfish-4.1 correspon a la versió 4.1. Heu de fer servir el que correspon a la vostra versió.

A continuació creeu un test que utilitzarà el recurs JDBC definit a UserServiceImpl i que, per tant, escriurà dades a la BD MySQL que heu creat.

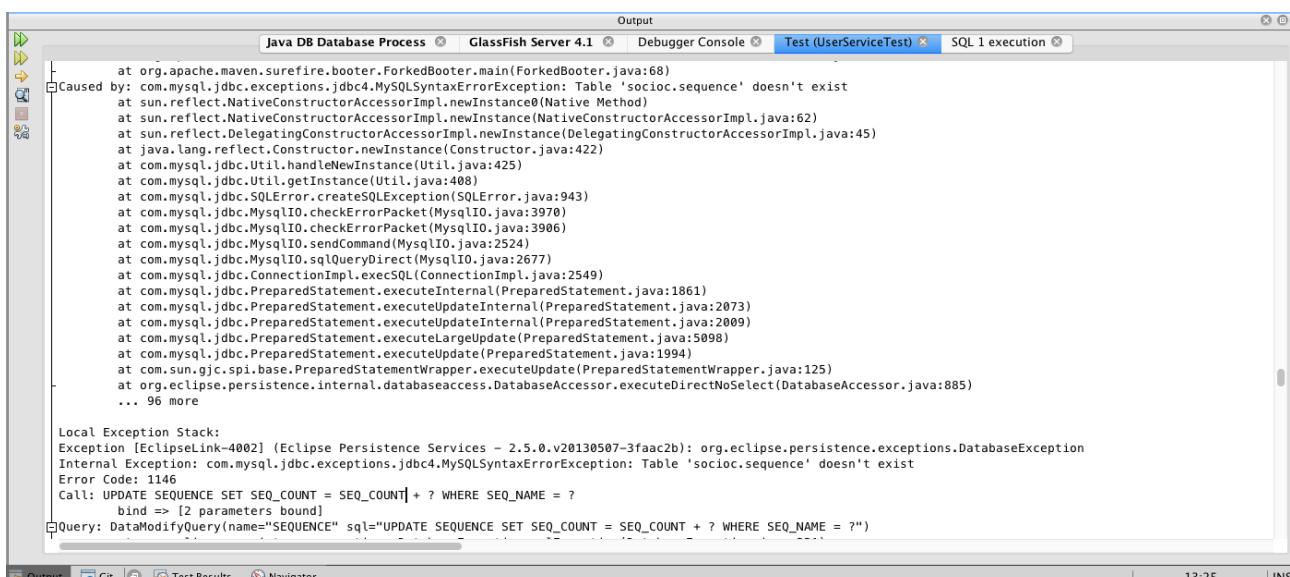
```

1  @Before
2  public void setUp() throws NamingException {
3      Context context = EJBContainer.createEJBContainer().getContext();
4      userService = (UserService) context.lookup("java:global/classes/
5          UserServiceImpl");
6  }
7
8  @Test
9  public void findUserByUsername() {
10     String username = "jdoe";
11     User user = new User();
12     user.setActive(true);
13     user.setCreatedOn(new Timestamp(new Date().getTime()));
14     user.setEmail("test@test.com");
15     user.setName("Jane");
16     user.setPassword("password");
17     user.setRank(100);
18     user.setUsername(username);
19     User user1 = new User();
20     user1.setActive(true);
21     user1.setCreatedOn(new Timestamp(new Date().getTime()));
22     user1.setEmail("test1@test.com");
23     user1.setName("Joe");
24     user1.setPassword("password");
25     user1.setRank(100);
26     user1.setUsername("joeTest");
27
28     userService.create(user);
29     userService.create(user1);
30
31     User userFromDB = userService.findUserByUsername(username);
32     Assert.assertNotNull(userFromDB);
33     Assert.assertEquals("jdoe", userFromDB.getUsername());
34     Assert.assertEquals("test@test.com", userFromDB.getEmail());
35     Assert.assertNotNull(userFromDB.getUserId());
}

```

Primer inicialitzeu el servidor Glassfish-embedded i recupereu l'EJB del contingidor d'EJB del servidor Glassfish real. Un cop teniu l'EJB UserServiceImpl ja podeu fer les operacions amb la base de dades. Executeu el test, i hauríeu d'obtenir un error similar al de la figura 2.24.

FIGURA 2.24. Error SQL



Per què creieu que teniu aquest error?

- Perquè no es pot connectar amb la BD?
- Perquè la taula “Users” no està disponible?
- Perquè la taula “Sequence” no existeix?

L'error és perquè la taula “Sequence” no existeix, no l'heu definit i no hi feu referència enllot; llavors, quan s'està intentant crear? Per què quan vau fer el test amb la BD en memòria no va fallar? La resposta està a la classe User.

```
1 @GeneratedValue
2   @Column(name = "user_id")
3   private Long userId;
```

L'anotació `GeneratedValue` té com a estratègia per defecte `GenerationType.AUTO`, que intentarà crear la taula “Sequence” per guardar el valor autogenerat per al camp “user_id” de la taula. Com que la taula no existeix, l'aplicació acaba amb un error, ja que no hi pot escriure. A la vostra taula vau definir que el cap que formaria la clau principal s'autoincrementaria, llavors l'estratègia que heu d'utilitzar és `.GenerationType.IDENTITY`. Modifiqueu la classe User.

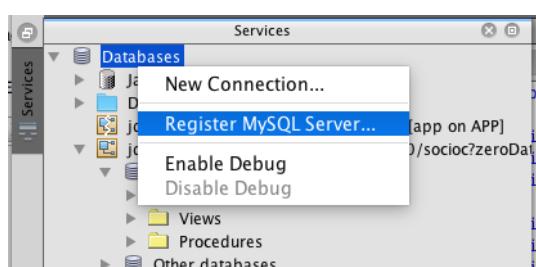
```
1 @GeneratedValue(strategy = GenerationType.IDENTITY)
2   @Column(name = "user_id")
3   private Long userId;
```

Executeu el test un altre cop, aquesta vegada tindreu el següent error:

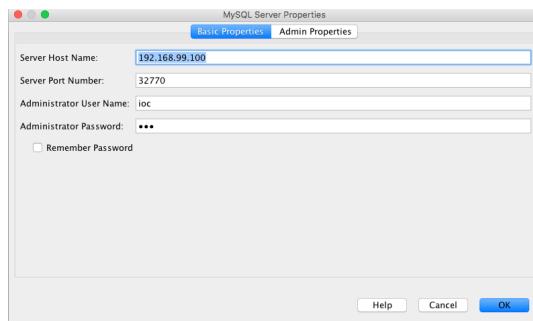
```
1 Caused by: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Unknown
column 'user_id' in 'field list'
```

L'error diu ara que no existeix un camp anomenat “user_id” a la taula “Users”. Netbeans deixa crear una connexió amb la BD que us permetrà examinar l'estructura de la taula. Seleccioneu la pestanya “Services”, feu clic amb el botó dret a *Database* i registreu una connexió amb un servidor MySQL, tal com podeu veure en la figura 2.25.

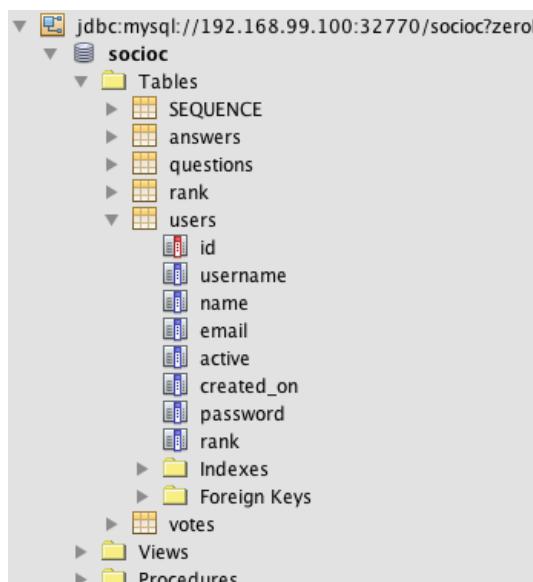
FIGURA 2.25. Registrar un servidor MySQL a Netbeans



A continuació, afegiu les dades del vostre servidor MySQL (vegeu la figura 2.26).

FIGURA 2.26. Registrar el servidor MySQL a Netbeans

Un cop definida la connexió ja podeu examinar el contingut de la taula “Users” (vegeu la figura 2.27) a través de la pestanya “Services” de Netbeans.

FIGURA 2.27. Contingut de la taula “Users”

El problema és que la columna de la taula “Users” s’anomena “id” i no “user_id”. Canviem, doncs, la classe User.

```

1 @Id
2   @GeneratedValue(strategy = GenerationType.IDENTITY)
3   @Column(name = "id")
4   private Long userId;

```

Executeu de nou el test, i ara acabarà sense cap error. Com que heu utilitzat la BD MySQL, podeu veure que els dos usuaris creats al tests estan a la taula “Users”. Amb l’opció *Execute command* (vegeu la figura 2.28) podeu executar consultes i veure que la taula conté els registres esperats (vegeu la figura 2.29).

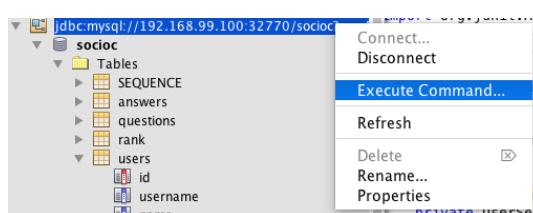
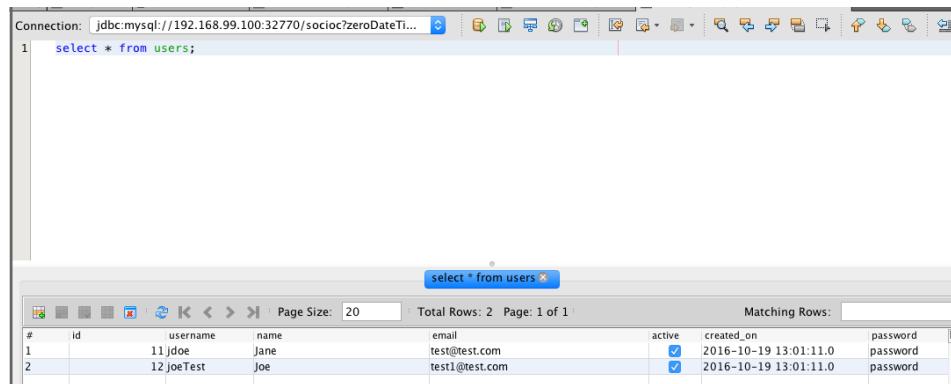
FIGURA 2.28. Executar una consulta SQL

FIGURA 2.29. Registres creats durant el test


The screenshot shows a MySQL Workbench interface. At the top, the connection string is `jdbc:mysql://192.168.99.100:32770/socio?zeroDateTi...`. Below it, a SQL editor window contains the query `select * from users;`. To the right of the editor is a results grid. The grid has columns: #, id, username, name, email, active, created_on, and password. There are two rows of data:

#	id	username	name	email	active	created_on	password
1	11jdoe	Jane		test@test.com	<input checked="" type="checkbox"/>	2016-10-19 13:01:11.0	password
2	12joeTest	Joe		test1@test.com	<input checked="" type="checkbox"/>	2016-10-19 13:01:11.0	password

2.2.4 Refactoritzant el codi per simplificar el codi i millorar el test

Escriure test unitaris que modifiquin el contingut de la base de dades MySQL és un problema. Penseu que aquesta serà la BD que utilitzareu per a la vostra aplicació; tenir un test que escriu dades a aquesta BD és, doncs, un error. Us ha servit per veure com utilitzar el servidor Glassfish per emprar un recurs JDBC i escriure a la BD, però ho heu de canviar.

Un altre problema està a la classe `UserServiceImpl`. El primer problema és que esteu determinant quina unitat de persistència utilitzarà aquesta EJB, i això impossibilita escollir quina unitat de persistència emprar quan s'executen els tests. Heu de refactoritzar el codi per tal de poder escollir la unitat de persistència en funció que executeu tests unitaris o desplegueu l'aplicació al servidor Glassfish.

```
1 @PersistenceContext(unitName = "MysqlResourceSocIocPersistence")
```

Un altra cosa que podeu millorar es la gestió de les transaccions. Aquest és el codi que heu utilitzat:

```
1 @Override
2 public void create(User user) {
3     UserTransaction utx = context.getUserTransaction();
4     try {
5         utx.begin();
6         entityManager.persist(user);
7         utx.commit();
8     } catch (Exception e) {
9         e.printStackTrace();
10    try {
11        utx.rollback();
12    } catch (Exception e1) {
13        e1.printStackTrace();
14    }
15 }
16 }
```

El que passa és que quan s'utilitza l'anotació `@Stateless`, automàticament EJB proporciona la següent anotació a cadascun dels mètodes `@TransactionAttribute(TransactionAttributeType.REQUIRED)`. Això fa que s'encarregui automàticament de gestionar les transaccions; per tant, podeu simplificar la classe `UserServiceImpl` notablement.

```

1  @Stateless
2  public class UserServiceImpl implements UserService {
3      @PersistenceContext
4      private EntityManager entityManager;
5
6      @Override
7      public void create(User user) {
8          entityManager.persist(user);
9      }
10
11     @Override
12     public void edit(User user) {
13         entityManager.merge(user);
14     }
15
16     @Override
17     public void remove(User user) {
18         entityManager.remove(user);
19     }
20
21     @Override
22     public User findUserByUsername(String username) {
23         return (User) entityManager.createQuery("select object(o) from User o " +
24             "where o.username = :username")
25             .setParameter("username", username)
26             .getSingleResult();
27     }
28 }
```

De la forma que heu fet el test anterior era necessari que el servidor Glassfish estigués corrent. Això és un problema, ja que necessiteu executar els tests de manera independent del servidor d'aplicacions, i el més important: heu de ser capaços de poder especificar la unitat de persistència que voleu utilitzar en cada moment. Per poder aconseguir això fareu servir Arquillian (red.ht/2ls9x1Q), que és un *framework* de testeig que facilita escriure tests del components EJB. Utilitzant Arquillian podreu llançar una versió d'un servidor Glassfish en memòria que us permetrà accedir a les EJB tal com si estiguessin corrent al servidor real. Per aconseguir això, el primer que heu de fer és afegir unes dependències. Hi ha uns quants canvis, per la qual cosa aquí teniu tot el fitxer pom.xml.

```

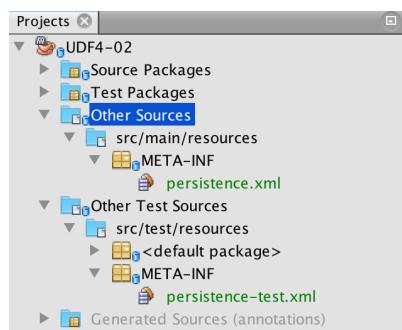
1 <dependencyManagement>
2     <dependencies>
3         <dependency>
4             <groupId>org.jboss.arquillian</groupId>
5             <artifactId>arquillian-bom</artifactId>
6             <version>1.1.11.Final</version>
7             <scope>import</scope>
8             <type>pom</type>
9         </dependency>
10    </dependencies>
11 </dependencyManagement>
12
13 <dependencies>
14     <dependency>
15         <groupId>com.h2database</groupId>
16         <artifactId>h2</artifactId>
17         <version>1.4.190</version>
18     </dependency>
19     <dependency>
20         <groupId>junit</groupId>
21         <artifactId>junit</artifactId>
22         <version>4.12</version>
23     </dependency>
24     <dependency>
```

```

25      <groupId>javax.validation</groupId>
26      <artifactId>validation-api</artifactId>
27      <version>1.1.0.Final</version>
28  </dependency>
29  <dependency>
30      <groupId>org.eclipse.persistence</groupId>
31      <artifactId>eclipselink</artifactId>
32      <version>2.6.4</version>
33  </dependency>
34  <dependency>
35      <groupId>javax.ejb</groupId>
36      <artifactId>javax.ejb-api</artifactId>
37      <version>3.2</version>
38  </dependency>
39  <dependency>
40      <groupId>mysql</groupId>
41      <artifactId>mysql-connector-java</artifactId>
42      <version>5.1.40</version>
43  </dependency>
44
45  <dependency>
46      <groupId>javax.el</groupId>
47      <artifactId>javax.el-api</artifactId>
48      <version>3.0.1-b04</version>
49  </dependency>
50
51  <dependency>
52      <groupId>org.jboss.arquillian.container</groupId>
53      <artifactId>arquillian-glassfish-embedded-3.1</artifactId>
54      <version>1.0.0.Final</version>
55      <scope>test</scope>
56  </dependency>
57  <dependency>
58      <groupId>org.glassfish.main.extras</groupId>
59      <artifactId>glassfish-embedded-all</artifactId>
60      <version>4.1.1</version>
61      <scope>provided</scope>
62  </dependency>
63  <dependency>
64      <groupId>javax.inject</groupId>
65      <artifactId>javax.inject</artifactId>
66      <version>1</version>
67  </dependency>
68
69  <dependency>
70      <groupId>org.jboss.arquillian.junit</groupId>
71      <artifactId>arquillian-junit-container</artifactId>
72      <scope>test</scope>
73  </dependency>
74 </dependencies>
```

El primer que fareu serà separar el fitxer on definiu les unitats de persistència en dos fitxers. Un contindrà la unitat de persistència per escriure a la BD MySQL i l'altre la unitat de persistència per a testeig (vegeu la figura 2.30).

FIGURA 2.30. Registres creats durant el test



Aquest és la unitat de persistència per escriure a MySQL:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5      http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
6      <persistence-unit name="SocIocPersistenceUnit" transaction-type="JTA">
7          <jta-data-source>jdbc/socioc_pm</jta-data-source>
8          <class>org.ioc.daw.user.User</class>
9          <properties>
10             <property name="eclipselink.logging.level" value="FINEST"/>
11         </properties>
12     </persistence-unit>
13 </persistence>
```

I aquest és la unitat de persistència que utilitzareu per al testeig:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5      http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
6      <persistence-unit name="SocIocPersistenceUnit-TEST" transaction-type="JTA">
7          <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
8          <class>org.ioc.daw.user.User</class>
9          <properties>
10             <property name="javax.persistence.jdbc.driver" value="org.h2.Driver
11                 "/>
12             <property name="javax.persistence.jdbc.user" value="username"/>
13             <property name="javax.persistence.jdbc.password" value="password"/>
14             <property name="javax.persistence.jdbc.url" value="
15                 jdbc:h2:mem:socioc_db;INIT=runscript from 'classpath:init.sql'
16                 "/>
17             <property name="javax.persistence.schema-generation.database.action
18                 " value="drop-and-create"/>
19             <property name="javax.persistence.schema-generation.create-source"
20                 value="metadata"/>
21             <property name="javax.persistence.schema-generation.drop-source"
22                 value="metadata"/>
23         </properties>
24     </persistence-unit>
25 </persistence>
```

Ara ja podeu reescriure el test unitari. Si teniu el servidor Glassfish en execució pareu-lo, o tindreu conflictes de port.

```

1  @RunWith(Arquillian.class)
2  public class UserServiceTest {
3      @Inject
```

```

4   private UserService userService;
5
6   @Deployment(testable = true)
7   public static JavaArchive createTestableDeployment() {
8       final JavaArchive jar = ShrinkWrap.create(JavaArchive.class, "example.
9           jar")
10      .addClasses(UserService.class, UserServiceImpl.class)
11      .addAsManifestResource("META-INF/persistence-test.xml", "
12          persistence.xml")
13      .addAsManifestResource(EmptyAsset.INSTANCE, ArchivePaths.create
14          ("beans.xml"));
15      return jar;
16  }
17
18  @Test
19  public void findUserByUsername() {
20      String username = "jdoe";
21      User user = new User();
22      user.setActive(true);
23      user.setCreatedOn(new Timestamp(new Date().getTime()));
24      user.setEmail("test@test.com");
25      user.setName("Jane");
26      user.setPassword("password");
27      user.setRank(100);
28      user.setUsername(username);
29      User user1 = new User();
30      user1.setActive(true);
31      user1.setCreatedOn(new Timestamp(new Date().getTime()));
32      user1.setEmail("test1@test.com");
33      user1.setName("Joe");
34      user1.setPassword("password");
35      user1.setRank(100);
36      user1.setUsername("joeTest");
37
38      userService.create(user);
39      userService.create(user1);
40
41      User userFromDB = userService.findUserByUsername(username);
42      Assert.assertNotNull(userFromDB);
43      Assert.assertEquals("jdoe", userFromDB.getUsername());
44      Assert.assertEquals("test@test.com", userFromDB.getEmail());
45      Assert.assertNotNull(userFromDB.getUserId());
46  }
47
48 }
```

`@RunWith(Arquillian.class)` indica que executareu el test utilitzant Arquillian, i a continuació injecteu l'EJB UserService. Això és possible perquè Arquillian crea un servidor Glassfish en memòria amb un contingut amb totes les EJB de la vostra aplicació, la qual cosa us permet afegir-les a una determinada classe quan sigui necessari. A `createTestableDeployment` és on es configura el servidor d'aplicacions utilitzant Arquillian. La part més important és on indiqueu quines classes són les EJB que voleu afegir al contingut EJB del servidor i qui és el fitxer amb la definició de la unitat de persistència que voleu fer servir.

Podeu trobar tot el codi resultant de la refactorització als annexos de la unitat.

2.3 Què s'ha après?

Heu après que utilitzant JDBC directament al codi té una sèrie d'inconvenients, ja que fa que us hagieu d'encarregar de programar operacions de baix nivell amb la base de dades, com establir i tancar les connexions, encarregar-se del mapatge de les dades de la BD amb els objectes del codi, etc.

Per solucionar aquests problemes heu vist JPA, que és una especificació que determina i fa estàndards les operacions amb la BD. És una especificació però no hi ha una implementació, així que per fer els exemples hem utilitzat EclipseLink, una de les implementacions disponibles. També heu vist que, encara que pugueu utilitzar implementacions de JPA per accedir directament a les bases de dades, utilitzar un servidor EJB com Glassfish proporciona diversos avantatges.

Pel que fa a JPA, permet establir un *pool* de connexions per fer més eficients les operacions i reutilitzar les connexions establertes amb la BD, que són operacions molt costoses. Finalment, heu vist diferents formes de testejar aplicacions que treballen amb bases de dades i heu creat una estructura, separant les unitats de persistència i utilitzant el *framework* Arquillian, que pot servir de base per testejar qualsevol aplicació que treballi amb JPA.

3. Accés a dades amb Spring i Hibernate

Java té una API que ens facilita la feina d'escriure codi que interactuï amb una base de dades. Aquesta API és la Java Persistence API (JPA), i és simplement una especificació que defineix les interfícies per fer operacions amb la BD. JPA no consisteix en cap implementació de les dades i per si sola no ens servirà de res. Necessitem unes llibreries que implementin aquesta especificació i que siguin capaces de transformar objectes Java en registres a la BD (ORM, Object-Relational Mapping) a més d'implementar totes les operacions amb la BD. Hi ha moltes implementacions, com ara Eclipselink (www.eclipse.org/eclipselink), Open JPA (openjpa.apache.org) o TopLink (bit.ly/2lAMG8o), però una de les més populars és Hibernate (hibernate.org).

Hibernate és, doncs, un *framework* ORM per a Java que té un gran rendiment i velocitat i facilita considerablement la feina de programació amb BD. Amb una senzilla configuració, permet establir una relació directa entre classes Java amb taules i tipus de dades SQL i facilita fins i tot la creació automàtica de les taules. Hibernate s'encarregarà aproximadament del 90% de la feina que s'ha de fer per treballar amb una BD automatitzant tasques repetitives. Una altra característica que fa de Hibernate un *framework* molt popular és el fet que té un llenguatge propi de consulta amb la BD que es diu Hibernate Query Language (HQL). Això permet utilitzar qualsevol de les 10 BD suportades per Hibernate sense haver de canviar ni una sola línia de codi, ja que Hibernate s'encarregarà de traduir les consultes HQL que fem a un llenguatge SQL específic per a cadascuna de les BD. Altres característiques són:

- És un projecte Opensource amb llicència LGPL.
- Alt rendiment: utilitza una memòria *cache* interna que fa que les operacions de lectura de la BD (normalment sempre hi ha moltes més operacions de lectura que d'escriptura) siguin molt ràpides.
- Creació automàtica de les taules.
- Simplifica l'obtenció de dades de múltiples taules.

Hibernate ens ajudarà pel que fa a les bases de dades, i Spring, per la seva banda, ens ajudarà en el disseny de la nostra aplicació. Spring és un *framework* que utilitza injecció de dependències (en anglès, DI, Dependency Injection) o la inversió del control (en anglès, IoC, Inversion of Control). IoC es refereix al fet que una classe no s'encarregarà de crear instàncies de les seves dependències, sinó que el contenidor DI s'encarregarà de crear els objectes amb la configuració necessària i injectar-los on faci falta. Aquest fet, que sembla tan simple, té grans implicacions a l'hora de desenvolupar una aplicació. Afavoreix la composició sobre l'erència de classes, la qual cosa fa que hi hagi menys dependències entre les classes. I

menys dependències implica que les classes faran menys coses i més específiques, per la qual cosa el codi serà més fàcilment reutilitzable i més fàcilment testeable.

La combinació de Spring i Hibernate ens permetrà dissenyar i desenvolupar un codi que pugui treballar fàcilment amb diferents tipus de BD (Hibernate) i on Spring ens donarà flexibilitat per canviar si és necessari Hibernate per qualsevol altre *framework* ORM.

Continuarem amb el desenvolupament de l'aplicació Java “SocLoc”. Explicarem:

- Hibernate
- Spring i IoC: inversió del control o injecció de dependències
- HQL
- Com configurar Spring i Hibernate
- Anotacions
- Relacions 1..M i N..M
- Validació
- Tests unitaris

3.1 "Socloc". Dialogant amb usuaris amb Spring i Hibernate

Les classes Java haurien de ser al més independents possible d'altres classes. Això augmenta la possibilitat de reutilitzar aquestes classes i simplifica els tests unitaris. Per aconseguir aquesta separació o desacoblamet, la dependència que una classe tingui amb d'altres s'ha d'injectar, més que fer que la classe creï o busqui la dependència. Això representa el principi d'inversió de control o principi de Hollywood: “no ens truquis” (crear/buscar objectes), “nosaltres et trucarem” (injectarem els objectes). Per exemple, la classe A depèndrà de B si usa la classe B com a variable. Si usem injecció de dependències, la classe B serà passada a la A via el constructor (injecció de construcció) o a través d'un mètode *setter* (injecció *setter*). Vegem un exemple:

```

1  public interface UserDAO {
2      public void create(User user);
3      public User edit(User user);
4      public void remove(User user);
5      public User findUserByUsername(String username);
6      public User findUserWithHighestRank();
7      public List<User> findActiveUsers();
8  }
9
10 @Stateless
11 public class UserDAOJPA implements UserDAO {
12     @PersistenceContext
13     private EntityManager entityManager;
14
15     @Override

```

```

16  public void create(User user) {
17      entityManager.persist(user);
18  }
19
20  @Override
21  public User edit(User user) {
22      return entityManager.merge(user);
23  }
24
25  @Override
26  public void remove(User user) {
27      user = entityManager.merge(user);
28      entityManager.remove(user);
29  }
30
31  @Override
32  public User findUserByUsername(String username) {
33      try {
34          return (User) entityManager.createQuery("select object(o) from User
35              o " +
36                  "where o.username = :username")
37                  .setParameter("username", username)
38                  .getSingleResult();
39      } catch (NoResultException e) {
40          return null;
41      }
42  }
43
44  @Override
45  public List<User> findActiveUsers() {
46      try {
47          return (List<User>) entityManager.createQuery("select object(o)
48              from User o " +
49                  "where o.active= true")
50                  .getResultList();
51      } catch (NoResultException e) {
52          return null;
53      }
54  }
55
56  @Override
57  public User findUserWithHighestRank() {
58      try {
59          return (User) entityManager.createQuery("select object(o) from User
60              o order by o.rank DESC")
61                  .setMaxResults(1)
62                  .getSingleResult();
63      } catch (NoResultException e) {
64          return null;
65      }
66  }
67
68  public class UserController {
69      private UserDAO userDAO;
70
71      public User getUser(String username){
72          return userDAO.findUserByUsername(username);
73      }
74 }
```

Podeu veure que la classe UserController utilitza la interfície UserDAO. Si us hi fixeu, el codi no fa referència a cap implementació de la interfície. Hem de modificar el codi de UserController per fer referència a una implementació.

```

1  public class UserService {
2      private UserDAO userDAO;
```

```

3
4     public UserService() {
5         this.userDAO = new UserDAOJPA();
6     }
7
8     public User getUser(String username) {
9         return userDAO.findUserByUsername(username);
10    }
11 }
```

En aquest cas, fem que la implementació de la interfície sigui la de la classe `UserDAOJPA`, que ens proporciona la funcionalitat per accedir a la BD utilitzant JPA. Aquest codi té diversos problemes. La classe `UserService` té la configuració de la implementació de la interfície `UserDAO`. Això farà que per testejar la classe `UserService` utilitzem un objecte real `UserDAOJPA`, que establirà una connexió amb una BD. Un altre problema és que si canviem la implementació de `UserDAO` i fem una implementació que usa Hibernate, haurem de venir a la classe `UserController` i canviar el codi. El que ens permet la injecció de dependències (DI) és que la classe `UserController` no coneixi els detalls de la configuració de `UserDAO`, sinó que aquests es passin. Si anéssim a utilitzar DI podríem refactoritzar el codi:

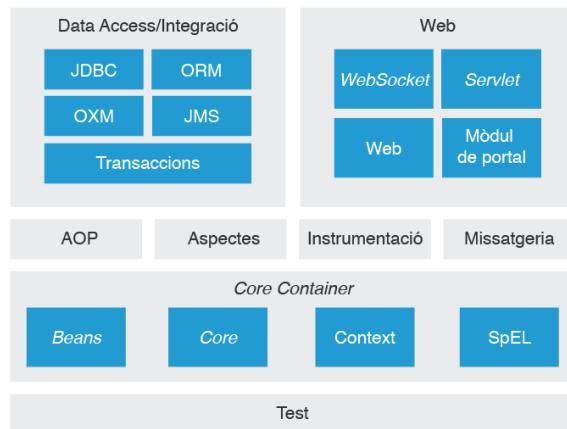
```

1  public class UserController {
2      private UserDAO userDAO;
3
4      public UserController(UserDAO userDAO) {
5          this.userDAO = userDAO;
6      }
7
8      public User getUser(String username){
9          return userDAO.findUserByUsername(username);
10     }
11 }
```

D'aquesta manera, la classe `UserController` està totalment desacoblada de la implementació de `UserDAO`. El *framework* de DI s'encarregarà de passar la versió correcta de `UserDAO` a la classe `UserController`. La injecció de dependències es pot aconseguir amb Java Standard. Spring, però, simplifica el procés mitjançant una forma estàndard de configurar les dependències entre els objectes.

3.1.1 Integrant l'aplicació "Socloc" amb Spring

Spring és un *framework* d'inversió del control (IoC) format d'una sèrie de mòduls que faciliten la feina de desenvolupar aplicacions Java. En ser modular, ens permet escollir quins mòduls utilitzar segons les necessitats de l'aplicació que estem desenvolupant. Hi ha uns 20 mòduls, i en la figura 3.1 podeu veure la seva arquitectura.

FIGURA 3.1. Mòduls Spring

Core container: aquest és el mòdul fonamental de Spring, i permet fer IoC i DI. També defineix el context de l'aplicació que indicarà quins *beans* (objectes gestionats pel *container* IoC) hi ha i com es relacionen entre si. Està format per quatre mòduls:

- Core proporciona les parts fonamentals del *framework*, incloent IoC i injecció de dependències.
- Beans proporciona BeanFactory, que és una classe que segueix el patró de disseny de *software* anomenat *factory pattern* i que serveix per crear objectes a partir d'una classe.
- Context proporciona la funcionalitat per accedir als objectes gestionats per Spring (*beans*). La interfície ApplicationContext interface és la part central d'aquest mòdul.
- SpEL proporciona un llenguatge potent i flexible per manipular un *bean* i les seves dependències mentre l'aplicació s'està executant.

Data Access/Integration: aquest és el mòdul que proporciona accés a dades i sistemes d'integració. Està format pels següents mòduls:

- JDBC proporciona la funcionalitat JDBC per accedir a bases de dades.
- ORM proporciona capes d'integració per a API de mapeig d'objectes Java amb taules de les BD. Entre les API suportades hi ha JPA, JDO, Hibernate i iBatis.
- OXM proporciona la funcionalitat per a la transformació d'objectes a XML, i viceversa.
- JMS (Java Messaging Service) és un mòdul que conté la funcionalitat per consumir i produir missatges JMS.
- Transaction és un mòdul que facilita la gestió de transaccions.

Web: la capa web està formada pels següents mòduls:

- Web proporciona funcionalitats típiques que s'utilitzen a les aplicacions web, com pujada de fitxers o la inicialització del contingut IoC utilitzant *servlets*.
- Web-MVC: conté una implementació de MVC (Model View Controller) per a aplicacions web.
- Web-Socket proporciona suport per a comunicacions client-servidor en aplicacions web.
- Web-Portlet proporciona una implementació de MVC per ser utilitzada en entorns *portlet*.

Hi ha altres mòduls importants:

- AOP (Aspect-Oriented Programming): consisteix en una implementació de programació orientada a aspectes. AOP permet interceptar crides a funcions i injectar codi que s'executarà abans o després d'executar el codi de la funció.
- Aspects: aquest mòdul permet la integració d'AspectJ, que és un *framework* d'AOP.
- Instrumentation: proporciona instrumentació de classes i funcionalitat per carregar classes que són necessàries a certs servidors d'aplicacions.
- Test: permet testejar aplicacions Spring utilitzant *frameworks* de testeig, com JUnit o TestNG.

No tots aquests mòduls seran necessaris quan desenvolupem una aplicació. El que serà fonamental serà afegir el *core*, que inclou el contingut IoC, que serà l'encarregat de crear instances dels objectes, configurar-los i crear les dependències necessàries. El contingut IoC obté aquesta informació de la configuració de Spring que es pot definir amb fitxers XML o amb classes de configuració Java.

L'arxiu de partida per treballar el descrit en aquest apartat el teniu disponible als annexos de la unitat.

El primer que haurem de fer és afegir les dependències. A partir del fitxer proporcionat als annexos afegirem les següents dependències:

```

1 <properties>
2   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3   <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
4   <springframework.version>4.3.4.RELEASE</springframework.version>
5   <mysql.connector.version>5.1.40</mysql.connector.version>
6   <junit.version>4.12</junit.version>
7   <mockito.version>1.10.19</mockito.version>
8   <h2.version>1.4.190</h2.version>
9 </properties>
10 <dependencies>
11 ...
12 <dependency>
13   <groupId>org.springframework</groupId>
14   <artifactId>spring-core</artifactId>
15   <version>${springframework.version}</version>
16 </dependency>
17 <dependency>
18   <groupId>org.springframework</groupId>
19   <artifactId>spring-web</artifactId>
```

```

20   <version>${springframework.version}</version>
21 </dependency>
22 <dependency>
23   <groupId>org.springframework</groupId>
24   <artifactId>spring-webmvc</artifactId>
25   <version>${springframework.version}</version>
26 </dependency>
27 <dependency>
28   <groupId>org.springframework</groupId>
29   <artifactId>spring-tx</artifactId>
30   <version>${springframework.version}</version>
31 </dependency>
32 <dependency>
33   <groupId>org.springframework</groupId>
34   <artifactId>spring-orm</artifactId>
35   <version>${springframework.version}</version>
36 </dependency>
37 <dependency>
38   <groupId>org.springframework</groupId>
39   <artifactId>spring-test</artifactId>
40   <version>${springframework.version}</version>
41   <scope>test</scope>
42 </dependency>
43 <dependency>
44   <groupId>org.mockito</groupId>
45   <artifactId>mockito-all</artifactId>
46   <version>${mockito.version}</version>
47   <scope>test</scope>
48 </dependency>
49 ...
50 </dependencies>
```

Fixeu-vos que hem definit una sèrie de propietats al pom.xml que permeten centralitzar la versió utilitzada en una variable. D'aquesta manera, quan hi hagi disponible una nova versió de Spring, enlloc de canviar el valor en sis dependències, només ho haurem de fer en un lloc.

```

1 <properties>
2   <springframework.version>4.3.4.RELEASE</springframework.version>
3 </properties>
```

A la implementació que tenim de UserService, la classe crea una instància de UserDAO. El que volem ara és que Spring s'encarregui d'aquesta configuració. Refactoritzarem UserService de manera que no creï una instància de UserDAO.

```

1 public class UserService {
2     private UserDAO userDAO;
3
4     public UserService(UserDAO userDAO) {
5         this.userDAO = userDAO;
6     }
7
8     public User getUser(String username) {
9         return userDAO.findUserByUsername(username);
10    }
11 }
```

El que farem serà que Spring s'encarregui d'injectar la configuració de UserDAO necessària. Això vol dir que podrem testejar la classe UserService sense necessitar tenir cap informació de com serà la implementació de UserDAO. Per poder fer-ho necessitem configurar la nostra aplicació per tal que utilitzi Spring. Spring permet escollir quin serà el context a utilitzar en els tests. El que nosaltres

volem testejar ara és que la classe `UserService` i les seves dependències estan gestionades per Spring. El que faci la implementació de la classe `UserDAO` realment no ens interessa en aquest moment, per la qual cosa utilitzarem un *mock* de la classe. Definim llavors la classe que tindrà la configuració del context de Spring a `src/test/java` al paquet package `org.ioc.daw.config`:

```

1 package org.ioc.daw.config;
2
3 import org.ioc.daw.user.UserDAO;
4 import org.ioc.daw.user.UserService;
5 import org.mockito.Mockito;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.context.annotation.Configuration
8
9 @Configuration
10 public class SpringTestConfig {
11     @Bean
12     public UserDAO userDAO() {
13         return Mockito.mock(UserDAO.class);
14     }
15
16     @Bean
17     public UserService userService(UserDAO userDAO) {
18         return new UserService(userDAO);
19     }
20 }
```

`@Configuration` indica que la classe conté un o més mètodes anotats amb `@Bean` i que produeixen *beans* gestionats pel contingidor de Spring. Aquesta configuració defineix dos *beans* que estaran gestionats pel contingidor IoC de Spring. Això permetrà injectar aquests dos *beans* quan ens faci falta.

Fixeu-vos que l'objecte que retorna `userDAO()` no és cap implementació real de la interfície. Utilitzant `Mockito.mock(UserDAO.class)` retornem un *mock*, és a dir, un objecte que fa de *proxy* amb la interfície però que no té cap codi. El que això permet és oblidar-se completament del funcionament de les classes que implementen `UserDAO` i focalitzar el test en `UserService`. Si l'objecte *mock* no té cap implementació, com es pot usar en el codi? Al test veureu que podeu definir què retornen els diferents mètodes quan són invocats. Creeu la classe de test `UserServiceTest` al paquet package `org.ioc.daw.user`.

Mockito és un *framework* de testeig que facilita la creació d'objectes de test (*mock*) que amaguen la implementació real i que faciliten els tests unitaris.

```

1 import org.ioc.daw.config.SpringTestConfig;
2 import org.junit.Test;
3 import org.junit.runner.RunWith;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.test.context.ContextConfiguration;
6 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
7
8 import static org.junit.Assert.assertEquals;
9 import static org.mockito.Mockito.times;
10 import static org.mockito.Mockito.verify;
11 import static org.mockito.Mockito.when;
12
13 @RunWith(SpringJUnit4ClassRunner.class)
14 @ContextConfiguration(classes = {SpringTestConfig.class})
15 public class UserServiceTest {
16     @Autowired
17     private UserDAO userDAO;
18
19     @Autowired
20     private UserService userService;
```

```

22  @Test
23  public void getUserByUsername() {
24      String username = "test";
25      User user = new User();
26      user.setUsername(username);
27      user.setId(1L);
28
29      when(userDAO.findUserByUsername(username)).thenReturn(user);
30
31      User userResult = userService.getUser(username);
32      assertEquals(username, userResult.getUsername());
33      assertEquals(new Long(1), userResult.getId());
34      verify(userDAO, times(1)).findUserByUsername(username);
35  }
36 }
```

@RunWith(SpringJUnit4ClassRunner.class) especifica que el test carregarà un context de Spring per ser utilitzat en un test JUnit, i @ContextConfiguration(classes = {SpringTestConfig.class}) especifica quines classes tenen la configuració del context. Com que hem definit que SpringTestConfig serà la configuració a utilitzar, podem injectar (amb la notació @Autowired) els *beans* definits. Fixeu-vos que com que UserDAO és un *mock*, podem dir el que retornaran els seus mètodes:

```
1 when(userService.findUserByUsername(username)).thenReturn(user);
```

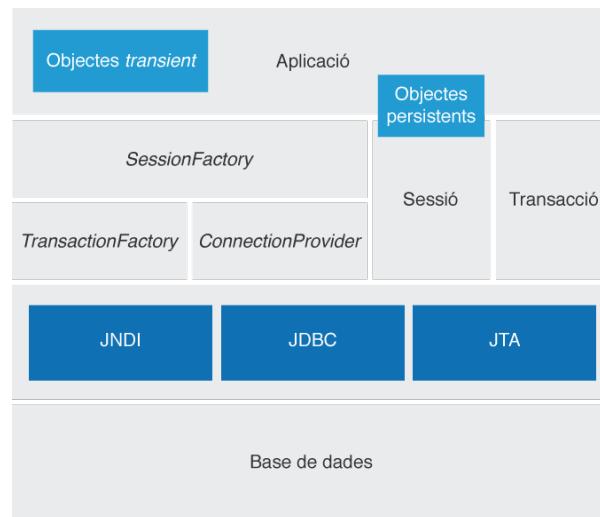
Estem dient que quan es cridi al mètode findUserByUsername retornarem l'objecte user. Les línies que utilitzen assertEquals fan les comprovacions del test, comproven que l'objecte retornat pel mètode findUserByUsername de UserService és el mateix que el que retorna UserDAO. Finalment, verify(userDAO, times(1)).findUserByUsername(username); comprova que el mètode findUserByUsername només és invocat un cop. Podeu trobar el codi en el fitxer .

3.1.2 Integrant l'aplicació "Socloc" amb Hibernate

Hibernate és un *framework* ORM (Object-Relational Mapping). És a dir, Hibernate s'encarrega de relacionar taules d'una base de dades amb objectes Java. Com es pot veure en la figura 3.2, Hibernate crea una capa entre la BD i l'aplicació. S'encarregarà de gestionar la configuració de com accedir a la BD, el tipus de BD, com fer el mapeig entre les classes i taules, i establir les relacions entre diferents taules.

FIGURA 3.2. Arquitectura d'Hibernate

En la figura 3.3 es representa amb més detall com funciona Hibernate. A l'hora de guardar les dades a la BD, Hibernate crea una instància de la classe de tipus entitat (una classe Java mapejada amb una taula). Aquest objecte s'anomena objecte *transient*, ja que no està associat amb cap sessió i no està guardat a la BD. Per guardar un objecte a la BD s'utilitza una instància de la interfície SessionFactory, un objecte de tipus *singleton* (només hi ha una instància de l'objecte a l'aplicació) que implementa el patró de disseny *factory*. SessionFactory carrega la configuració d'Hibernate i s'encarrega de gestionar la configuració de la connexió amb la BD.

FIGURA 3.3. Arquitectura detallada d'Hibernate

Cada connexió amb la BD a Hibernate es fa creant una instància d'una implemetació de la interfície Session. Hibernate també disposa d'una API per gestionar les transaccions i que permet utilitzar transaccions JDBC o JTA. Una transacció representa una única unitat de treball amb la base de dades. Vegem amb una mica més de detall els diferents blocs de la figura 3.3:

- SessionFactory: és una classe encarregada de produir objectes de tipus Session. Opcionalment, manté una memòria *cache* de segon nivell que guarda dades de la connexió amb la BD perquè siguin reutilitzades entre diferents transaccions.

- Session: s'encarreguen de la conversa entre les aplicacions i la BD. Manté una *cache* de primer nivell dels objectes de l'aplicació. Aquesta *cache* s'utilitza a l'hora de recuperar objectes utilitzant el seu identificador o a l'hora de navegar a través de les dependències de l'objecte.
- Objectes persistents: són objectes que contenen la funcionalitat de l'aplicació. Cada objecte està associat amb una única sessió d'Hibernate. Un cop la sessió associada a un objecte es tanca, els objectes passen a estar a l'estat *detached* (separat).
- Objectes tipus *transient* i *detached*: són les instàncies de classes de tipus Entity que no estan associades a cap sessió d'Hibernate. Poden haver estat creades per l'aplicació i no haver estat guardades, o poden ser el resultat que s'hagi tancat una sessió d'Hibernate.
- Proveïdor de connexions (ConnectionProvider): és opcional i permet crear un *pool* de connexions JDBC.
- TransactionFactory: permet crear instàncies d'objectes de tipus Transaction.

Per poder treballar amb Hibernate i que formi part de la vostra aplicació, el primer que fareu serà afegir les dependències necessàries. Importeu a Netbeans el codi descarregat dels annexos. Modificareu el fitxer pom.xml per afegir les dependències d'Hibernate.

```

1 <properties>
2   ...
3     <hibernate.version>5.2.5.Final</hibernate.version>
4   ...
5 </properties>
6 <dependency>
7   <groupId>org.hibernate</groupId>
8   <artifactId>hibernate-validator</artifactId>
9   <version>5.3.4.Final</version>
10 </dependency>
11 <dependency>
12   <groupId>org.hibernate</groupId>
13   <artifactId>hibernate-core</artifactId>
14   <version>${hibernate.version}</version>
15 </dependency>
16 <dependency>
17   <groupId>org.jadira.usertype</groupId>
18   <artifactId>usertype.core</artifactId>
19   <version>6.0.1.GA</version>
20 </dependency>
```

Als annexos de la unitat trobareu un arxiu amb el codi per importar a Netbeans i treballar amb Hibernate.

Hibernate a la llibreria *hibernate-core* porta l'especificació JPA 2.1 i la seva implementació. Per tant, no és necessari incloure les següents dependències.

```

1 <dependency>
2   <groupId>javax.persistence</groupId>
3   <artifactId>persistence-api</artifactId>
4   <version>1.0.2</version>
5 </dependency>
```

Hibernate utilitza el concepte de sessions per gestionar les connexions amb la base de dades. A cada sessió s'obre una única connexió amb la BD i s'utilitza

fins que la sessió es tanca. Cada objecte que es carrega en memòria per Hibernate estarà associat amb la sessió. Això permet a Hibernate persistir automàticament els objectes que s'han modificat. Quan la sessió persisteix canvis a la BD es diu *flushing*. Cada objecte associat amb la sessió es comprova per veure si ha canviat d'estat. Qualsevol objecte amb canvi d'estat es conservarà a la base de dades, amb independència que els objectes modificats es guardin o no explícitament. Aquesta característica es pot configurar, però per defecte podeu configurar el comportament arran d'Hibernate, es farà automàticament. Hibernate fa *flushing* en les següents situacions:

- Quan s'executa directament el mètode `flush()`.
- Abans que Hibernate faci una consulta, si creu que és necessari per obtenir un resultat precís.
- Quan es confirma una transacció.
- Quan es tanca la sessió.

El que volem fer a continuació és començar a utilitzar Hibernate. Per fer-ho definireu una classe que implementi la interfície UserDAO i que utilitzi la funcionalitat d'Hibernate. Creareu la classe `org.ioc.daw.user.UserHibernateDAO`.

```

1  import org.hibernate.Criteria;
2  import org.hibernate.Session;
3  import org.hibernate.SessionFactory;
4  import org.hibernate.criterion.Order;
5  import org.hibernate.criterion.Restrictions;
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.stereotype.Repository;
8  import javax.transaction.Transactional;
9  import java.util.List;
10
11 @Transactional
12 @Repository("userHibernateDAO")
13 public class UserHibernateDAO implements UserDAO {
14
15     @Autowired
16     private SessionFactory sessionFactory;
17
18     @Override
19     public void create(User user) {
20         getSession().saveOrUpdate(user);
21     }
22
23     @Override
24     public User edit(User user) {
25         return (User) getSession().merge(user);
26     }
27
28     @Override
29     public void remove(User user) {
30         getSession().delete(user);
31     }
32
33     @Override
34     public User findUserByUsername(String username) {
35         Criteria criteria = createEntityCriteria();
36         criteria.add(Restrictions.eq("username", username));
37         return (User) criteria.uniqueResult();
38     }
39

```

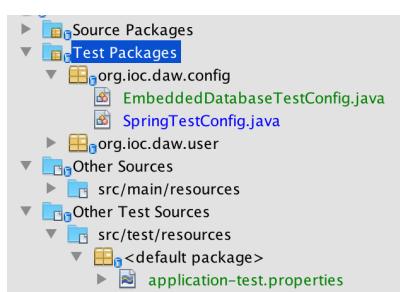
```

40
41     @Override
42     public User findUserWithHighestRank() {
43         Criteria criteria = createEntityCriteria();
44         criteria.addOrder(Order.desc("rank"));
45         return (User) criteria.uniqueResult();
46     }
47
48     @Override
49     public List<User> findActiveUsers() {
50         Criteria criteria = createEntityCriteria();
51         criteria.add(Restrictions.eq("active", true));
52         return (List<User>) criteria.list();
53     }
54
55     protected Session getSession() {
56         return sessionFactory.getCurrentSession();
57     }
58
59     private Criteria createEntityCriteria() {
60         return getSession().createCriteria(User.class);
61     }
}

```

@Repository("userHibernateDAO") indica que quan la classe sigui escanejada per Spring es crearà un *bean* anomenat userHibernateDAO. La notació @Transactional indica que els mètodes definits a la classe utilitzaran transaccions, és a dir, que quan el mètode es comença a executar s'obre una transacció i abans d'acabar es tanca. A les línies 4-5 injecteu l'objecte que permetrà fer totes les operacions utilitzant Hibernate SessionFactory i obtenir una sessió d'Hibernate. A la classe UserHibernateDAO heu vist com injectar les dependències necessàries per utilitzar Hibernate i com fer que els mètodes siguin transaccionals. El que heu de fer ara és crear la configuració que creï el *bean* SessionFactory, un altre *bean* que defineixi la classe que gestioni les transaccions, i finalment necessitareu definir un *bean* de tipus DataSource, que establirà com es farà la connexió amb la base de dades. El que interessa és testejar que Hibernate treballi amb la BD correctament, no tant la BD en si, i per això utilitzareu una BD en memòria. En aquest cas, H2. Al directori *test/java* creareu la classe org.ioc.daw.config.EmbeddedDatabaseTestConfig, i el fitxer de propietats al directori *test/resources application-test.properties*, tal com podeu veure en la figura 3.4.

FIGURA 3.4. Hibernate



```

1 import org.hibernate.SessionFactory;
2 import org.springframework.beans.factory.annotation.Autowired;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.context.annotation.PropertySource;
6 import org.springframework.core.env.Environment;

```

```

7   import org.springframework.jdbc.datasource.embedded.EmbeddedDatabase;
8   import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
9   import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
10  import org.springframework.orm.hibernate5.HibernateTransactionManager;
11  import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
12  import org.springframework.transaction.annotation.EnableTransactionManagement;
13  import javax.sql.DataSource;
14  import java.util.Properties;

15
16 @Configuration
17 @EnableTransactionManagement
18 @PropertySource(value = {"application-test.properties"})
19 public class EmbeddedDatabaseTestConfig {
20
21     @Autowired
22     private Environment environment;
23
24     @Bean
25     public UserDAO userDAO() {
26         return new UserHibernateDAO();
27     }
28
29     @Bean
30     public DataSource dataSource() {
31         EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
32         EmbeddedDatabase db = builder.setType(EmbeddedDatabaseType.H2).build();
33         return db;
34     }
35
36     @Bean
37     @Autowired
38     public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
39         LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
40         sessionFactory.setDataSource(dataSource);
41         sessionFactory.setPackagesToScan("org.ioc.daw");
42         sessionFactory.setHibernateProperties(hibernateProperties());
43         return sessionFactory;
44     }
45
46     private Properties hibernateProperties() {
47         Properties properties = new Properties();
48         properties.put("hibernate.dialect", environment.getRequiredProperty("hibernate.dialect"));
49         properties.put("hibernate.hbm2ddl.auto", environment.getRequiredProperty("hibernate.hbm2ddl"));
50         properties.put("hibernate.show_sql", environment.getRequiredProperty("hibernate.show_sql"));
51         properties.put("hibernate.format_sql", environment.getRequiredProperty("hibernate.format_sql"));
52         return properties;
53     }
54
55     @Bean
56     @Autowired
57     public HibernateTransactionManager transactionManager(SessionFactory s) {
58         HibernateTransactionManager txManager = new HibernateTransactionManager();
59         txManager.setSessionFactory(s);
60         return txManager;
61     }
62 }
```

`@EnableTransactionManagement` habilita la capacitat de gestionar les transaccions amb la BD. `@PropertySource(value = {"classpath:application-test.properties"})` permet definir propietats a un fitxer de propietats que serà accessible a través del component injectat *Environment*.

```

1 @Bean
2     public UserDAO userDAO() {
3         return new UserHibernateDAO();
4     }

```

Defineix que com el Bean de tipus UserDAO que utilitzarem serà del tipus UserHibernateDAO.

```

1 @Bean
2     public DataSource dataSource() {
3         EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
4         EmbeddedDatabase db = builder.setType(EmbeddedDatabaseType.H2).build();
5         return db;
6     }

```

Aquest codi crea un *bean* de tipus DataSource, que és el que establirà amb quina base de dades es connectarà Hibernate. En el vostre cas, indiqueu que serà una BD en memòria de tipus H2. Això serà l'únic que haureu de canviar en el codi si voleu utilitzar una BD diferent.

```

1 @Bean
2     @Autowired
3     public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
4         LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
5         sessionFactory.setDataSource(dataSource);
6         sessionFactory.setPackagesToScan("org.ioc.daw");
7         sessionFactory.setHibernateProperties(hibernateProperties());
8         return sessionFactory;
9     }

```

El mètode sessionFactory() crea un *bean* de tipus LocalSessionFactoryBean que té informació de com connectar amb la base de dades a través de l'objecte DataSource. Fixeu-vos que el bean DataSource s'injecta utilitzant la notació @Autowired. A més de DataSource, es necessita definir les propietats d'Hibernate (hibernateProperties()). Gràcies a la notació @PropertySource es poden externalitzar les propietats a fitxers, fet que permet carregar diferents fitxers de propietats a diferents contextos. Un cop l'objecte SessionFactory s'ha creat s'injectarà al mètode transactionManager, que proporcionarà suport per a les transaccions amb la base de dades.

```

1 @Bean
2     @Autowired
3     public HibernateTransactionManager transactionManager(SessionFactory s) {
4         HibernateTransactionManager txManager = new HibernateTransactionManager()
5             ;
6         txManager.setSessionFactory(s);
7         return txManager;
}

```

És a dir, Spring crearà tres *beans*: un que té informació de la BD a utilitzar, un que usa aquest *bean* per crear una sessió que gestiona la connexió amb la BD i un altre que gestiona les transaccions. El fitxer de propietats application-test.properties d'Hibernate té el contingut mostrat a continuació. A més de definir que el dialecte SQL que usarà Hibernate és H2Dialect, estem indicant també amb la propietat hibernate.hbm2ddl que es creïn les taules automàticament a partir dels objectes.

```

1 hibernate.dialect = org.hibernate.dialect.H2Dialect
2 hibernate.hbm2ddl = create
3 hibernate.show_sql = true
4 hibernate.format_sql = true

```

Un cop Hibernate i Spring estan configurats podem passar a crear el test UserDAOTest al paquet org.ioc.daw.user. De moment comprobareu que podeu escriure i llegir informació de la BD.

```

1 import org.ioc.daw.config.EmbeddedDatabaseTestConfig;
2 import static org.junit.Assert.*;
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.test.context.ContextConfiguration;
7 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
8 import java.sql.Timestamp;
9 import java.util.Date;

10
11
12 @RunWith(SpringJUnit4ClassRunner.class)
13 @ContextConfiguration(classes = {EmbeddedDatabaseTestConfig.class})
14 public class UserDAOTest {
15     @Autowired
16     private UserDAO userDAO;
17
18     @Test
19     public void saveUser() {
20         User user = new User();
21         user.setUsername("test");
22         user.setActive(true);
23         user.setEmail("email@test.com");
24         user.setPassword("password");
25         user.setName("name");
26         user.setRank(10);
27         user.setCreatedOn(new Timestamp(new Date().getTime()));
28         assertNull(user.getUserId());
29         userDAO.create(user);
30         assertNotNull(user.getUserId());
31
32         User userFromDb = userDAO.findUserByUsername("test");
33         assertEquals(user.getUserId(), userFromDb.getUserId());
34     }
35 }

```

Amb @ContextConfiguration(classes = {EmbeddedDatabaseTestConfig.class}) definiu quin serà el fitxer de propietats que utilitzareu al test; en el vostre cas és el fitxer de configuració que defineix un DataSource per connectar a la BD H2. Fixeu-vos que abans de guardar l'usuari el valor del seu userId és *null*, però que un cop Hibernate persisteix l'objecte, tal com es va definir a la classe User (@GeneratedValue(strategy = GenerationType.IDENTITY)), es genera un valor automàticament. El test consisteix a guardar l'usuari a la BD, recuperar-lo usant el nom d'usuari i comprovar que el userId dels dos objectes és el mateix.

Una de les característiques d'Hibernate és que un cop un objecte forma part d'una sessió si es fan canvis sobre alguna de les seves propietats, els canvis es persisteixen a la BD. Modificareu el test anterior per comprovar-ho.

```

1 @Test
2 public void saveUser(){

```

```

3 User user = new User();
4 user.setUsername("test");
5 user.setActive(true);
6 user.setEmail("email@test.com");
7 user.setPassword("password");
8 user.setName("name");
9 user.setRank(10);
10 user.setCreatedOn(new Timestamp(new Date().getTime()));
11
12 assertNull(user.getUserId());
13 userDAO.create(user);
14 assertNotNull(user.getUserId());
15 user.setEmail("new-email@test.com");
16
17 User userFromDb = userDAO.findUserByUsername("test");
18 assertEquals(user.getUserId(), userFromDb.getUserId());
19 assertEquals("new-email@test.com", userFromDb.getEmail());
20 }
```

Un cop s'ha guardat l'objecte user, modifiqueu el correu-e i comproveu que s'ha guardat correctament. Si executeu el test veureu que hi ha un error:

```
1 Failed tests: saveUser(org.ioc.daw.user.UserDAOTest): expected:<[new-]
email@test.com> but was:<[]email@test.com>
```

El problema és que necessiteu establir el context per a la transacció, és a dir, indicar on comença i on acaba la transacció. Si afegiu la notació @Transactional al mètode del test i torneu a executar el test veureu que aquest cop s'executa correctament.

Com podríeu utilitzar el test que heu creat per comprovar que podeu escriure dades a la BD MySQL “SociLoc”?

La configuració està definida a la classe `EmbeddedDatabaseTestConfig`, així que tot el que haureu de canviar estarà aquí:

Podeu accedir al codi al fitxer que trobareu als annexos de la unitat.

- `@PropertySource(value = {"classpath:application-test.properties"})` indica que s'han d'agafar els valors d'aquest fitxer de propietats per fer la connexió amb la BD.
- El bean `DataSource` indica que utilitzareu una BD en memòria.

Llavors, per connectar-vos a la BD MySQL només haureu de canviar el fitxer de propietats i el tipus de `DataSource`. Si feu el canvi a `EmbeddedDatabaseTestConfig`, si voleu tornar a utilitzar la BD en memòria, com serà el cas, haureu de tornar a canviar un altre cop el fitxer. Creareu una altra classe de configuració al paquet `org.ioc.daw.config` que establirà com treballar amb la BD MySQL que anomenarem `HibernateMysqlConfiguration`.

```

1 package org.ioc.daw.config;
2
3 import org.hibernate.SessionFactory;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.ComponentScan;
7 import org.springframework.context.annotation.Configuration;
8 import org.springframework.context.annotation.PropertySource;
9 import org.springframework.core.env.Environment;
```

```

10 import org.springframework.jdbc.datasource.DriverManagerDataSource;
11 import org.springframework.orm.hibernate5.HibernateTransactionManager;
12 import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
13 import org.springframework.transaction.annotation.EnableTransactionManagement;
14 import javax.naming.NamingException;
15 import javax.sql.DataSource;
16 import java.util.Properties;

17
18 @Configuration
19 @EnableTransactionManagement
20 @ComponentScan({"org.ioc.daw.user", "org.ioc.daw.question",
21                 "org.ioc.daw.answer", "org.ioc.daw.vote", "org.ioc.daw.rank"})
22 @PropertySource(value = {"jdbc.properties", "hibernate.properties"})
23 @Import(value = {HibernateConfiguration.class})
24 public class HibernateMysqlConfiguration {
25     @Autowired
26     private Environment environment;
27
28     @Bean
29     public DataSource dataSource() {
30         DriverManagerDataSource dataSource = new DriverManagerDataSource();
31         dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.
32             driverClassName"));
32         dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
33         dataSource.setUsername(environment.getRequiredProperty("jdbc.username")
34             );
34         dataSource.setPassword(environment.getRequiredProperty("jdbc.password")
35             );
36         return dataSource;
37     }
38 }
```

Afegiu també un fitxer de propietats src/main/resources/application.properties. Fixeu-vos que haureu de canviar jdbc.url per la IP:PORT on estigui funcionant la vostra base de dades.

```

1 jdbc.driverClassName = com.mysql.jdbc.Driver
2 jdbc.url = jdbc:mysql://192.168.99.100:32768/socioc?useUnicode=true&
3           useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&
4           serverTimezone=UTC
5 jdbc.username = root
6 jdbc.password = root
7 hibernate.dialect = org.hibernate.dialect.MySQLDialect
8 hibernate.show_sql = true
9 hibernate.format_sql = true
10 hibernate.hbm2ddl = validate
```

A causa de possibles problemes amb la configuració de MySQL i la seva configuració horària, afegiu una sèrie de paràmetres (?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC/) a la cadena de caracters de la connexió amb el servidor MySQL.

Si us hi fixeu, la classe de configuració HibernateMysqlConfiguration té repetit gairebé tot el codi respecte a EmbeddedDatabaseTestConfig. Per tant, si voleu fer algun canvi al DataSource o introduir una nova propietat d'Hibernate us haureu de recordar de canviar les dues classes. Això no és una bona pràctica de desenvolupament i és susceptible a errors. El que fareu serà separar les classes en tres: una classe de configuració que contindrà el codi comú i dues amb les configuracions específiques per a MySQL i H2, i afegir els DAO al seu propi fitxer de configuració. Creareu DAOConfig al paquet org.ioc.daw.config.

```

1 import org.ioc.daw.user.UserDAO;
2 import org.ioc.daw.user.UserHibernateDAO;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6 @Configuration
7 public class DAOConfig {
8
9     @Bean
10    public UserDAO userDAO() {
11        return new UserHibernateDAO();
12    }
13}

```

HibernateConfiguration conté la configuració comú:

```

1 import org.hibernate.SessionFactory;
2 import org.springframework.beans.factory.annotation.Autowired;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.core.env.Environment;
6 import org.springframework.orm.hibernate5.HibernateTransactionManager;
7 import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
8 import org.springframework.transaction.annotation.EnableTransactionManagement;
9 import javax.naming.NamingException;
10 import javax.sql.DataSource;
11 import java.util.Properties;
12 @Configuration
13 @EnableTransactionManagement
14 @Import(value = {DAOConfig.class})
15 public class HibernateConfiguration {
16     @Autowired
17     private Environment environment;
18
19     @Bean
20     @Autowired
21     public LocalSessionFactoryBean sessionFactory(DataSource dataSource) throws
22         NamingException {
23         LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
24         sessionFactory.setDataSource(dataSource);
25         sessionFactory.setPackagesToScan("org.ioc.daw.user");
26         sessionFactory.setHibernateProperties(hibernateProperties());
27         return sessionFactory;
28     }
29
30     private Properties hibernateProperties() {
31         Properties properties = new Properties();
32         properties.put("hibernate.dialect", environment.getRequiredProperty("-
33             hibernate.dialect"));
34         properties.put("hibernate.show_sql", environment.getRequiredProperty("-
35             hibernate.show_sql"));
36         properties.put("hibernate.format_sql", environment.getRequiredProperty(
37             "hibernate.format_sql"));
38         properties.put("hibernate.hbm2ddl.auto", environment.
39             getRequiredProperty("hibernate.hbm2ddl"));
40         return properties;
41     }
42
43     @Bean
44     @Autowired
45     public HibernateTransactionManager transactionManager(SessionFactory s) {
46         HibernateTransactionManager txManager = new HibernateTransactionManager()
47             ();
48         txManager.setSessionFactory(s);
49         return txManager;
50     }
51 }

```

HibernateMysqlConfiguration tindrà la configuració específica per connectar amb la BD MySQL.

```

1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.context.annotation.Bean;
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.context.annotation.Import;
5 import org.springframework.context.annotation.PropertySource;
6 import org.springframework.core.env.Environment;
7 import org.springframework.jdbc.datasource.DriverManagerDataSource;
8 import org.springframework.transaction.annotation.EnableTransactionManagement;
9 import javax.sql.DataSource;
10 @Configuration
11 @EnableTransactionManagement
12 @PropertySource(value = {"application.properties"})
13 @Import(value = {HibernateConfiguration.class})
14 public class HibernateMysqlConfiguration {
15     @Autowired
16     private Environment environment;
17
18     @Bean
19     public DataSource dataSource() {
20         DriverManagerDataSource dataSource = new DriverManagerDataSource();
21         dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.
22             driverClassName"));
23         dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
24         dataSource.setUsername(environment.getRequiredProperty("jdbc.username")
25             );
26         dataSource.setPassword(environment.getRequiredProperty("jdbc.password")
27             );
28         return dataSource;
29     }
30 }
```

I finalment, EmbeddedDatabaseTestConfig tindrà la configuració per connectar-se a la BD en memòria H2.

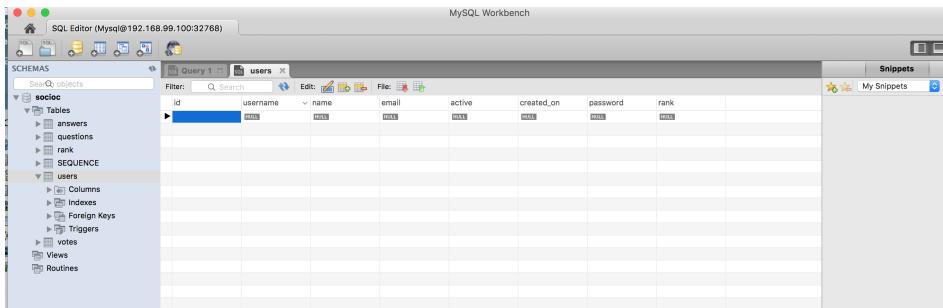
```

1 @Configuration
2 @EnableTransactionManagement
3 @PropertySource(value = {"application-test.properties"})
4 @Import(value = {HibernateConfiguration.class})
5 public class EmbeddedDatabaseTestConfig {
6
7     @Bean
8     public DataSource dataSource() {
9         EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
10        EmbeddedDatabase db = builder.setType(EmbeddedDatabaseType.H2).build();
11        return db;
12    }
13 }
```

Als annexos de la unitat trobareu un arxiu amb el codi refactoritzat, així com un altre arxiu de MySQL Workbench amb la base de dades “Socloc” per importar-la.

Comproveu que amb el codi refactoritzat podeu executar tots els tests. A continuació modificareu EmbeddedDatabaseTestConfig per tal d'executar els tests contra la BD MySQL. Podeu importar la base de dades “SocIoc” del fitxer de MySQL Workbench descarregat dels annexos.

Per comprovar que el vostre codi i configuració pot escriure a la BD MySQL que vau definir a Workbench, podeu utilitzar l'editor de *queries* de Workbench per fer la comprovació. En la figura 3.3 podeu veure que el contingut de la taula està buit.

FIGURA 3.5. Contingut de la taula “Users”

A continuació canvieu el context del test `UserDAOTest` perquè utilitzi la classe de configuració de MySQL i executeu el test `UserDAOTest.saveUser`.

```
1 @ContextConfiguration(classes = {HibernateMysqlConfiguration.class})
```

Si executeu el test hi haurà un altre error:

```
1 .HibernateConfiguration: Invocation of init method failed; nested exception is
  org.hibernate.tool.schema.spi.SchemaManagementException: Schema-validation
    : wrong column type encountered in column [id] in table [users]; found [
  int (Types#INTEGER)], but expecting [bigint (Types#BIGINT)]
```

El problema és que vau definir la taula “Users” com a tipus `int`, i Hibernate el que espera és un tipus `bigint`. Això passa perquè heu definit `userId` com a `Long`, que Hibernate mapeja amb el tipus `bigint` per tal d’acomodar a tots els possibles valors que pot guardar una variable de tipus `Long`. Per solucionar el problema modifiqueu la classe `User` canviant el tipus de `userId` de `Long` a `Integer`.

```
1 @Id
2   @NotNull
3   @GeneratedValue(strategy = GenerationType.IDENTITY)
4   @Column(name = "id")
5   private Integer userId;
6   public Integer getUserId() {
7     return userId;
8   }
9
10  public void setUserId(Integer userId) {
11    this.userId = userId;
12 }
```

Ara sí, executeu el test i mireu el contingut de la taula “Users”; veureu està buida. Què ha passat? Doncs que com que és un test, Hibernate fa un *rollback* (desfer els canvis d’una transacció) abans de tancar el test i esborra les dades guardades a la BD. Per tal que es guardin les dades a la BD és necessari indicar al test que no faci *rollback* amb la notació `@Rollback(false)`. El mètode amb el test tindrà llavors les següents anotacions:

```
1 @Test
2   @Transactional
3   @Rollback(false)
4   public void saveUser() {
```

Ara sí que veureu que hi ha l’usuari que el test ha creat (vegeu la figura 3.4).

FIGURA 3.6. Contingut de la taula “Users”

	id	active	created_on	email	name	password	rank	username
	49	1	2016-12-19 1...	new-email@te...	10	test

Aquests canvis que heu fet són només per veure que podeu guardar dades a la BD només canviant un fitxer de configuració. Desfeu els canvis (treure la notació @Rollback i utilitzar la configuració EmbeddedDatabaseTestConfig) abans de continuar endavant.

A continuació veurem com podreu configurar Spring i Hibernate per treballar amb el servidor d'aplicacions Glassfish. Primer afegireu al fitxer pom.xml una dependència que us permetrà cercar recursos (en el vostre cas, un recurs JDBC) utilitzant JNDI. Java Naming and Directory Interface és una API que permet trobar recursos, serveis i components EJB que estan distribuïts. Afegiu el següent al pom.xml:

```

1 <dependency>
2   <groupId>org.glassfish.main.common</groupId>
3   <artifactId>glassfish-naming</artifactId>
4   <version>4.1.1</version>
5 </dependency>
```

A continuació heu de crear una classe de configuració amb els detalls específics de Glassfish al paquet org.ioc.daw.config. Fixeu-vos que l'únic que heu d'especificar és el tipus de DataSource. En aquest cas utilitzeu JndiDataSourceLookup per buscar el recurs JDBC especificat per a la propietat jndi.socioc del fitxer de propietats glassfish-application.properties.

```

1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.context.annotation.Bean;
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.context.annotation.Import;
5 import org.springframework.context.annotation.PropertySource;
6 import org.springframework.core.env.Environment;
7 import org.springframework.jdbc.datasource.lookup.JndiDataSourceLookup;
8 import org.springframework.transaction.annotation.EnableTransactionManagement;
9 import javax.sql.DataSource;
10
11 @Configuration
12 @EnableTransactionManagement
13 @PropertySource(value = {"glassfish-application.properties"})
14 @Import(value = {HibernateConfiguration.class})
15
16 public class GlassfishPoolConfiguration {
17
18     @Autowired
19     private Environment environment;
20
21     @Bean(name = "jndiDataSource")
22     public DataSource dataSource() {
23         String jndiName = environment.getRequiredProperty("jndi.socioc");
24         JndiDataSourceLookup lookup = new JndiDataSourceLookup();
25         lookup.setResourceRef(true);
26         return lookup.getDataSource(jndiName);
27     }
}
```

28 }

El fitxer de propietats només ha de contenir les propietats que configuren Hibernate i la propietat que indica el nom del recurs JNDI.

```
1 jndi.socioc = jdbc/socioc
2 hibernate.dialect = org.hibernate.dialect.MySQLDialect
3 hibernate.show_sql = true
4 hibernate.format_sql = true
5 hibernate.hbm2ddl = validate
```

Si us hi fixeu, les propietats d'Hibernate són les mateixes que les que vau utilitzar per a la connexió MySQL. Com que no voleu tenir codi ni configuració repetides, refactoritzareu els fitxers de propietats. Creareu el fitxer hibernate.properties amb les propietats d'Hibernate, jdbc.properties amb les propietats específiques per a JDBC i glassfish.properties amb les propietats de Glassfish.

hibernate.properties:

```
1 hibernate.dialect = org.hibernate.dialect.MySQLDialect
2 hibernate.show_sql = true
3 hibernate.format_sql = true
4 hibernate.hbm2ddl = validate
```

jdbc.properties:

```
1 jdbc.driverClassName = com.mysql.jdbc.Driver
2 jdbc.url = jdbc:mysql://192.168.99.100:32768/socioc
3 jdbc.username = root
4 jdbc.password = root
```

glassfish.properties:

```
1 jndi.socioc = jdbc/socioc
```

A continuació us heu d'assegurar que les classes de configuració inclouen els fitxers de propietats adients.

```
1 @PropertySource(value = {"glassfish.properties", "hibernate.properties"})
2 @Import(value = {HibernateConfiguration.class})
3 public class GlassfishPoolConfiguration {
4
5     @PropertySource(value = {"jdbc.properties", "hibernate.properties"})
6     @Import(value = {HibernateConfiguration.class})
7     public class HibernateMysqlConfiguration {
```

Finalment, voleu testejar que la nova configuració funciona i que podeu escriure a la BD MySQL. El problema és que el *bean* de tipus DataSource configurat per utilitzar el recurs JDBC a Glassfish, si l'aplicació no està desplegada al servidor Glassfish, no el trobarà. Hi diverses alternatives: una podria ser utilitzar un servidor Glassfish que s'executi com a part del test utilitzant Glassfish-embedded. Un altra aproximació, que és molt més flexible, ràpida i que ens permet testejar la funcionalitat de Spring, Hibernate i la nostra aplicació, és ampliar la funcionalitat de la classe Spring que executa els tests JUnit perquè sigui ella la que s'encarregui que el test tingui disponibles recursos JNDI. La idea és molt senzilla: crear un

context JNDI on registrareu un *bean* que després utilitzareu al test. Això ho podeu fer amb la classe org.ioc.daw.SpringJNDIRunner, que formarà part dels tests.

```

1 package org.ioc.daw;
2
3 import javax.naming.NamingException;
4 import org.ioc.daw.config.HibernateMysqlConfiguration;
5 import org.junit.runners.model.InitializationError;
6 import org.springframework.context.ApplicationContext;
7 import org.springframework.context.annotation.
8     AnnotationConfigApplicationContext;
9 import org.springframework.mock.jndi.SimpleNamingContextBuilder;
10 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
11
12 public class SpringJNDIRunner extends SpringJUnit4ClassRunner {
13     public static boolean isJNDIactive;
14     public SpringJNDIRunner(Class<?> klass) throws InitializationError,
15         IllegalStateException, NamingException {
16         super(klass);
17
18         synchronized (SpringJNDIRunner.class) {
19             if (!isJNDIactive) {
20
21                 ApplicationContext applicationContext = new
22                     AnnotationConfigApplicationContext(
23                         HibernateMysqlConfiguration.class);
24                 SimpleNamingContextBuilder builder = new
25                     SimpleNamingContextBuilder();
26                 builder.bind("jdbc/socioc", applicationContext.getBean("
27                     dataSource"));
28                 builder.activate();
29
30             isJNDIactive = true;
31         }
32     }
33 }
34 }
```

Amb new AnnotationConfigApplicationContext definiu un context de Spring que carrega el *bean* definit a la classe de configuració HibernateMysqlConfiguration. A continuació registreu el *bean* anomenat dataSource amb el nom JNDI jdbc/socioc. El *bean* dataSource és el definit a HibernateMysqlConfiguration, que permet la connexió amb la BD MySQL. Per executar el test només heu d'especificar la nova classe que utilitzareu per definir el context de Spring en el qual correrà el test i el fitxer on heu configurat el DataSource que busca el recurs JDBC utilitzant JNDI.

```

1 @RunWith(SpringJNDIRunner.class)
2 @ContextConfiguration(classes = {GlassfishPoolConfiguration.class})
```

Assegureu-vos de buidar el contingut de la taula "Users" abans d'executar el test. Aquest és el problema de treballar amb BD reals amb els tests, que mai es pot saber en quin es troba la BD i quines dades conté abans d'executar els tests. Això pot fer que els tests fallin, no per un error en el codi, sinó per un error en les dades.

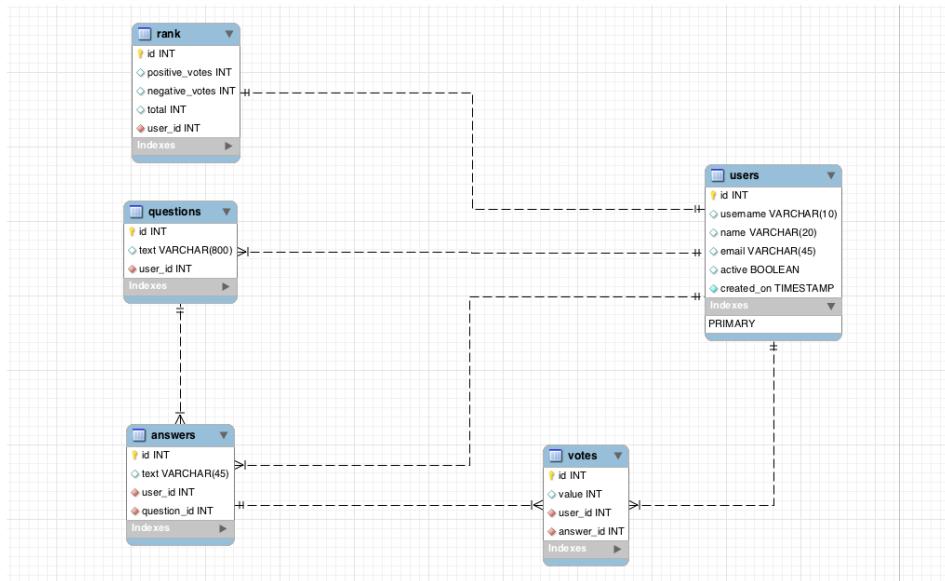
Podeu trobar el codi al fitxer que teniu disponible als annexos de la unitat.

Heu vist els avantatges d'utilitzar Spring i Hibernate, així com una breu introducció al seu funcionament. Després heu vist com configurar i integrar l'aplicació "Socloc" amb aquests dos frameworks i com treballar amb diferents recursos JDBC, ja sigui en memòria, connectant directament amb MySQL o utilitzant Glassfish i l'accés als seus recursos utilitzant JNDI.

3.2 "Socloc". Dialogant amb preguntes i respostes

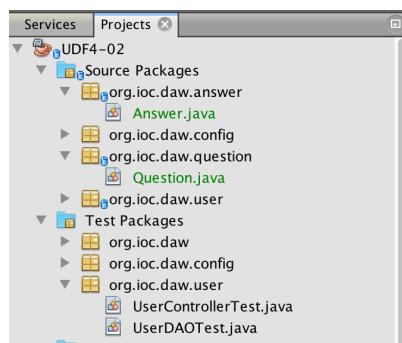
A continuació aprofundirem més a fons en Hibernate i explorar les relacions entre els objectes i com es guardaran en la BD. Per fer-ho treballarem amb les preguntes, respostes i vots de l'aplicació SocLoc. Recordeu les taules de la base de dades “SocLoc” i com es relacionen. En la figura 3.7 podeu veure que les preguntes (taula “Questions”) poden tenir més d’una resposta (taula “Answers”), i que un usuari pot fer més d’una pregunta i contestar moltes altres.

FIGURA 3.7. Contingut de la taula “Users”



Començareu creant les classes que representen les preguntes i respostes (vegeu la figura 3.8).

FIGURA 3.8. Classes Question i Answer



```

1 import javax.persistence.Column;
2 import javax.persistence.Entity;
3 import javax.persistence.GeneratedValue;
4 import javax.persistence.GenerationType;
5 import javax.persistence.Id;
6 import javax.persistence.Table;
7 import javax.validation.constraints.NotNull;
8 import javax.validation.constraints.Size;
9 import java.io.Serializable;
  
```

```
10  @Entity
11  @Table(name = "answers")
12  public class Answer implements Serializable {
13      private static final long serialVersionUID = 1L;
14      @Id
15      @NotNull
16      @GeneratedValue(strategy = GenerationType.IDENTITY)
17      @Column(name = "id")
18      private Integer answerId;
19
20      @NotNull
21      @Size(max = 45)
22      @Column(name = "text")
23      private String text;
24
25      public Integer getAnswerId() {
26          return answerId;
27      }
28
29      public void setAnswerId(Integer answerId) {
30          this.answerId = answerId;
31      }
32
33      public String getText() {
34          return text;
35      }
36
37      public void setText(String text) {
38          this.text = text;
39      }
40  }
41
42  import org.ioc.daw.answer.Answer;
43  import javax.persistence.CascadeType;
44  import javax.persistence.Column;
45  import javax.persistence.Entity;
46  import javax.persistence.FetchType;
47  import javax.persistence.GeneratedValue;
48  import javax.persistence.GenerationType;
49  import javax.persistence.Id;
50  import javax.persistence.OneToMany;
51  import javax.persistence.Table;
52  import javax.validation.constraints.NotNull;
53  import javax.validation.constraints.Size;
54  import java.io.Serializable;
55  import java.util.Set;
56  @Entity
57  @Table(name = "questions")
58  public class Question implements Serializable {
59      private static final long serialVersionUID = 1L;
60      @Id
61      @NotNull
62      @GeneratedValue(strategy = GenerationType.IDENTITY)
63      @Column(name = "id")
64      private Integer questionId;
65
66      @NotNull
67      @Size(max = 800)
68      @Column(name = "text")
69      private String text;
70
71      @OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
72      private Set<Answer> answers;
73
74      public Integer getQuestionId() {
75          return questionId;
76      }
77
78      public void setQuestionId(Integer questionId) {
79          this.questionId = questionId;
```

```

80 }
81
82     public String getText() {
83         return text;
84     }
85
86     public void setText(String text) {
87         this.text = text;
88     }
89
90     public Set<Answer> getAnswers() {
91         return answers;
92     }
93
94     public void setAnswers(Set<Answer> answers) {
95         this.answers = answers;
96     }
97 }
```

Com podeu veure, a la classe Question s'ha definit la relació amb les respostes amb la notació @OneToMany, que indica que una pregunta podrà tenir moltes respostes. CascadeType.ALL indica que si volem persistir un objecte que té com a atribut algun objecte que no està persistit, i per tant gestionat per Hibernate, el persistirem. Per exemple, si voleu guardar un objecte User que té una pregunta que no està guardada, la guardarà. De la mateixa manera, si esborreu un usuari que té preguntes, totes les preguntes s'esborraran. FetchType indica l'estrategia que s'utilitzarà per carregar les dades, FetchType.EAGER recuperarà les dades immediatament i FetchType.LAZY ho farà quan faci falta. S'ha de tenir present que per poder utilitzar FetchType.LAZY la sessió ha d'estar encara oberta quan s'intenti accedir a les dades. En el vostre cas FetchType.EAGER és suficient, però s'ha d'estudiar cada cas per veure quina estratègia de càrrega de dades s'utilitza.

```

1 @OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
2     private Set<Answer> answers;
```

Fixeu-vos que a la classe Question no hi ha cap referència als usuaris; com es fa llavors per indicar la relació? Les preguntes no tenen usuaris, una pregunta estarà formulada per un usuari, però serà l'entitat de tipus usuari la que pot tenir múltiples preguntes. Definiu aquesta relació a la classe User. De la mateixa manera, afegiu la relació amb Answer.

```

1 @OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
2     private Set<Question> questions;
3
4     @OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
5     private Set<Answer> answers;
6
7     public Set<Question> getQuestions() {
8         return questions;
9     }
10
11    public void setQuestions(Set<Question> questions) {
12        this.questions = questions;
13    }
14
15    public Set<Answer> getAnswers() {
16        return answers;
17    }
18
19    public void setAnswers(Set<Answer> answers) {
20        this.answers = answers;
```

```
21 }
```

Un cop definides les noves entitats, heu de canviar la configuració de `HibernateConfiguration` per tal que Hibernate escanegi els nous paquets.

```
1 public class HibernateConfiguration {
2     ....
3     @Bean
4     @Autowired
5     public LocalSessionFactoryBean sessionFactory(DataSource dataSource) throws
6         NamingException {
7         LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
8         sessionFactory.setDataSource(dataSource);
9         sessionFactory.setPackagesToScan("org.ioc.daw.user", "org.ioc.daw.
10             question", "org.ioc.daw.answer");
11         sessionFactory.setHibernateProperties(hibernateProperties());
12     }
13     ....
```

Un cop teniu definides les relacions, definireu els objectes per accedir a les dades (DAO). Creeu `org.ioc.daw.question.QuestionDAO` i la seva implementació `org.ioc.daw.question.QuestionHibernateDAO`.

```
1 public interface QuestionDAO {
2     Question getById(Integer questionId);
3     void save(Question question);
4     Question update(Question question);
5 }
6
7 import org.hibernate.Session;
8 import org.hibernate.SessionFactory;
9 import org.springframework.beans.factory.annotation.Autowired;
10 import org.springframework.stereotype.Repository;
11 import javax.transaction.Transactional;
12
13 @Transactional
14 @Repository("questionHibernateDAO")
15 public class QuestionHibernateDAO implements QuestionDAO {
16
17     @Autowired
18     private SessionFactory sessionFactory;
19
20     @Override
21     public Question getById(Integer questionId) {
22         return getSession().get(Question.class, questionId);
23     }
24
25     @Override
26     public void save(Question question) {
27         getSession().saveOrUpdate(question);
28     }
29
30     @Override
31     public Question update(Question question) {
32         return (Question) getSession().merge(question);
33     }
34
35     protected Session getSession() {
36         return sessionFactory.getCurrentSession();
37     }
38 }
```

I declareu el nou bean a `org.ioc.daw.config.DAOConfig`.

```

1 @Bean
2     public QuestionDAO questionDAO(){
3         return new QuestionHibernateDAO();
4     }

```

Amb aquest DAO que només permet guardar i trobar preguntes pel seu identificador, com podreu trobar per exemple totes les preguntes d'un usuari o afegir una resposta a una pregunta? Com que els objectes estan relacionats, per fer alguna d'aquestes operacions utilitzareu una combinació de diferents DAO. Creareu, al paquet `org.ioc.daw.question`, la interície `QuestionService` i la seva implementació `QuestionServiceImpl`, però primer afegireu un mètode a `UserDAO` que permeti obtenir un usuari utilitzant el seu *id*.

```

1 public interface UserDAO {
2     ...
3
4     public User getById(Integer id);
5     ...
6 }
7 public class UserDaoJPA implements UserDAO {
8     ....
9
10    @Override
11    public User getById(Integer id) {
12        try {
13            return (User) entityManager.createQuery("select object(o) from User
14                o " +
15                    "where o.id = :id")
16                    .setParameter("id", id)
17                    .getSingleResult();
18        } catch (NoResultException e) {
19            return null;
20        }
21    }
22    ....
23 }
24 @Repository("userHibernateDAO")
25 public class UserHibernateDAO implements UserDAO {
26     ....
27     @Override
28     public User getById(Integer userId) {
29         return getSession().get(User.class, userId);
30     }
31     ....
32 }

```

```

1 public interface QuestionService {
2     public Set<Question> getAllQuestions(Integer userId);
3     public void addAnswer(Answer answer, Integer questionId, Integer userId);
4     public void create(Question question, Integer userId);
5 }
6
7 import org.ioc.daw.answer.Answer;
8 import org.ioc.daw.user.User;
9 import org.ioc.daw.user.UserDAO;
10 import org.springframework.beans.factory.annotation.Autowired;
11 import javax.transaction.Transactional;
12 import java.util.HashSet;
13 import java.util.Set;
14
15 @Transactional
16 public class QuestionServiceImpl implements QuestionService {
17     @Autowired
18     private UserDAO userDAO;

```

```

19
20     @Autowired
21     private QuestionDAO questionDAO;
22
23
24     @Override
25     public Set<Question> getAllQuestions(Integer userId) {
26         User user = userDAO.getById(userId);
27         return user.getQuestions();
28     }
29
30     @Override
31     public Question addAnswer(Answer answer, Integer questionId, Integer userId
32         ) {
33         User user = userDAO.getById(userId);
34         Set<Answer> userAnswers = user.getAnswers();
35         addAnswerToCollection(answer, userAnswers);
36         user.setAnswers(userAnswers);
37         userDAO.create(user);
38
39         Question question = questionDAO.getById(questionId);
40         Set<Answer> answers = question.getAnswers();
41         answers = addAnswerToCollection(answer, answers);
42         question.setAnswers(answers);
43         return questionDAO.update(question);
44     }
45
46     private Set<Answer> addAnswerToCollection(Answer answer, Set<Answer>
47         answers) {
48         if (answers != null) {
49             answers.add(answer);
50         } else {
51             answers = new HashSet<Answer>();
52             answers.add(answer);
53         }
54         return answers;
55     }
56
57     @Override
58     public void create(Question question, Integer userId) {
59         User user = userDAO.getById(userId);
60         Set<Question> questions = user.getQuestions();
61         if (questions != null) {
62             questions.add(question);
63         } else {
64             questions = new HashSet<>();
65             questions.add(question);
66             user.setQuestions(questions);
67         }
68     }

```

Creeu un nou fitxer de configuracions org.ioc.daw.config.ServicesConfig i declareu el *now bean*.

```

1  @Configuration
2  public class ServicesConfig {
3      @Bean
4      public QuestionService questionService(){
5          return new QuestionServiceImpl();
6      }
7      @Bean
8      public UserService userService(UserDAO userDAO) {
9          return new UserService(userDAO);
10     }
11 }

```

Comproveu que estem injectant dependències de forma diferent als *beans* QuestionService i UserController. A UserController esteu injectant UserDao al constructor, mentre que a QuestionService ho feu amb la notació @Autowired, que injectarà els *beans* utilitzant els *setters*. Què és millor, utilitzar injecció per a *setters* o per a constructors? No és qüestió de millor ni pitjor, i realment depèn de cada cas. De forma general, si les dependències que esteu injectant són imprescindibles per al funcionament de la classe es recomana utilitzar injecció mitjançant constructors; per tant, refactoritzarem la classe QuestionServiceImpl. La classe que configura els *beans* de servei quedarà de la següent forma:

```

1  @Configuration
2  public class ServicesConfig {
3      @Bean
4      public QuestionService questionService(UserDAO userDao, QuestionDAO
5          questionDAO) {
6          return new QuestionServiceImpl(userDAO, questionDAO);
7      }
8
9      @Bean
10     public UserService userService(UserDAO userDao) {
11         return new UserService(userDAO);
12     }
13 }
```

```

1  public class QuestionServiceImpl implements QuestionService {
2      private UserDAO userDao;
3      private QuestionDAO questionDAO;
4
5      public QuestionServiceImpl(UserDAO userDao, QuestionDAO questionDAO) {
6          this.userDAO = userDao;
7          this.questionDAO = questionDAO;
8      }
9 }
```

A continuació creareu el test QuestionDAOTest per comprovar que les preguntes es guarden correctament a la base de dades.

```

1  import org.ioc.daw.config.EmbeddedDatabaseTestConfig;
2  import org.ioc.daw.config.ServicesConfig;
3  import org.ioc.daw.user.User;
4  import org.ioc.daw.user.UserDAO;
5  import org.junit.Test;
6  import org.junit.runner.RunWith;
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.test.context.ContextConfiguration;
9  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
10 import java.sql.Timestamp;
11 import java.util.Date;
12 import java.util.Set;
13 import static org.junit.Assert.assertEquals;
14 import static org.junit.Assert.assertNotNull;
15
16 @RunWith(SpringJUnit4ClassRunner.class)
17 @ContextConfiguration(classes = {ServicesConfig.class,
18     EmbeddedDatabaseTestConfig.class})
19 public class QuestionDAOTest {
20     @Autowired
21     private QuestionService questionService;
22
23     @Autowired
24     private UserDAO userDao;
25
26     @Test
```

```

26     public void createQuestion() {
27         User user = new User();
28         user.setUsername("test");
29         user.setActive(true);
30         user.setEmail("email@test.com");
31         user.setPassword("password");
32         user.setName("name");
33         user.setRank(10);
34         user.setCreatedOn(new Timestamp(new Date().getTime()));
35         userDAO.create(user);
36         Question question = new Question();
37         question.setText("This is a question");
38
39         questionService.create(question, user.getUserId());
40         assertNotNull(question.getQuestionId());
41
42         Set<Question> questions = questionService.getAllQuestions(user.
43             getUserId());
44         assertEquals(1, questions.size());
45     }

```

Podeu trobar aquest codi al fitxer que teniu disponible als annexos de la unitat.

A continuació comprobareu si la vostra aplicació seria capaç de guardar les dades a la base de dades “Socioc” que vau crear a MySQL. Per fer-ho només fa falta canviar el ContextConfiguration del test.

```

1 @ContextConfiguration(classes = {ServicesConfig.class,
    HibernateMysqlConfiguration.class})

```

Si executeu el test veureu el següent error on s'indica que la taula “questions_answers” no existeix.

```

1 Caused by: org.springframework.beans.factory.BeanCreationException: Error
   creating bean with name 'sessionFactory' defined in org.ioc.daw.config.
   HibernateConfiguration: Invocation of init method failed; nested exception
   is org.hibernate.tool.schema.spi.SchemaManagementException: Schema-
   validation: missing table [users_answers]

```

D'on surt aquesta taula? Per defecte, Hibernate utilitza taules intermèdies per desnormalitzar la base de dades. Això no té gaire a veure amb Hibernate *per se*, però sí amb el correcte disseny de la BD. En la taula 3.1 podeu veure com quedarien les dades de la taula “Questions” amb l'exemple d'un usuari que hagués fet diverses preguntes.

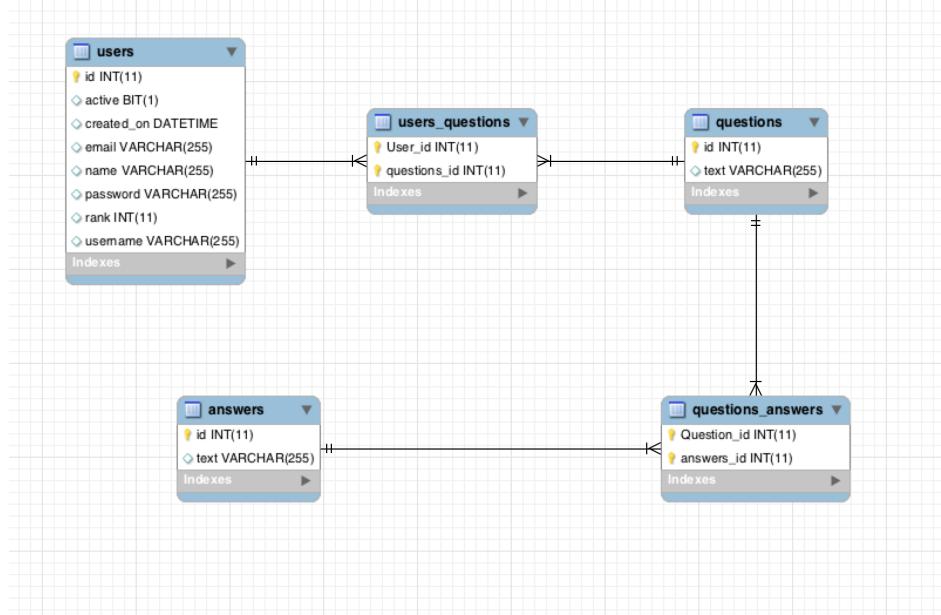
TAULA 3.1. Taula “Questions”

id	text	user_id
1	pregunta 1	2
2	pregunta 2	2
3	pregunta 3	4
4	pregunta 4	2

Podeu veure que el fet que la taula tingui el camp “user_id” fa que no representi només les dades d'una pregunta, sinó que hi ha també informació dels usuaris. Això trenca la segona norma de normalització de les base de dades, que diu que totes les columnes han de ser dependents de la clau principal. És a dir, si a la pregunta “aquesta columna descriu el que la clau principal identifica?” la resposta és no, llavors vol dir que la columna no és dependent de la clau principal. En el vostre cas, “la columna ‘user_id’ descriu el que la clau principal ‘id’ identifica (una

pregunta)?"", clarament no. Això implica que aquesta taula no està normalitzada. En la figura 3.9 podeu veure com es podria aconseguir la normalització per a les taules "Questions", "Users" i "Answers".

FIGURA 3.9. Normalització de les taules



Amb les noves taules, les dades que relacionen preguntes i usuaris quedarien tal com es pot veure en la taula 3.2 i la taula 3.3.

TAULA 3.2. Taula "Users questions"

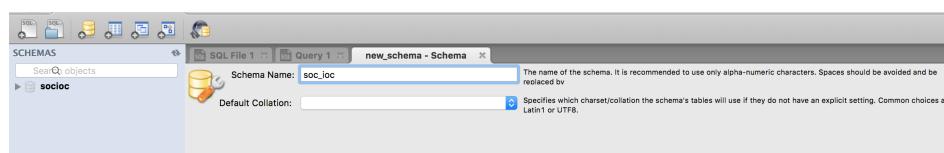
user_id	question_id
2	1
2	2
4	3
2	4

TAULA 3.3. Taula "Questions"

id	text
1	pregunta 1
2	pregunta 2
3	pregunta 3
4	pregunta 4

Un dels beneficis d'utilitzar Hibernate és que es pot encarregar d'això. El que fareu serà crear un nou esquema a MySQL que utilitzareu per fer que Hibernate s'encarregui de la generació de les taules necessàries. Creeu un nou esquema utilitzant MySQL Workbench anomenat "soc_ioc", tal com es mostra en la figura 3.10.

FIGURA 3.10. Schema soc_ioc



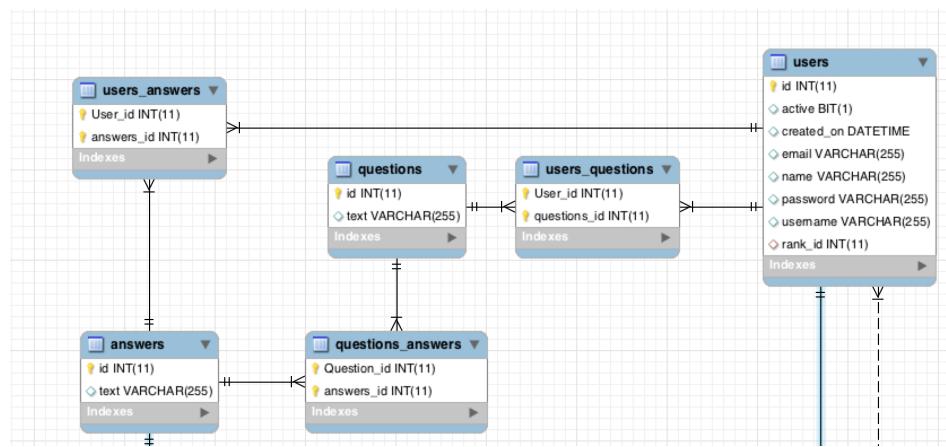
Canviu l'URL de connexió JDBC per utilitzar el nou esquema modificant el paràmetre `jdbc.url` del fitxer `jdbc.properties` i modifiqueu la propietat `hibernate.hbm2ddl` de `hibernate.properties` perquè creï les taules automàticament, però que no modifiqui les taules si hi ha canvis a les entitats.

```
1 jdbc.url = jdbc:mysql://192.168.99.100:32768/soc_ioc
```

```
1 hibernate.hbm2ddl = update
```

Si ara executeu el test `org.ioc.daw.question.QuestionDAOTest#createQuestion` comprobareu que no hi ha cap error. Si comproveu a MySQL Worbench veureu que Hibernate haurà creat les taules a partir de les entitats que hem definit, tal com podeu veure en la figura 3.11.

FIGURA 3.11. Taules creades per Hibernate



Ara que sabeu que el codi és capaç d'escriure i llegir dades correctament a la BD MySQL, tornareu a canviar la BD utilitzada als tests per a la BD en memòria H2 i hi afegireu més tests.

```

1 @Test
2     public void addAnswer() {
3         User user = new User();
4         user.setUsername("test");
5         user.setActive(true);
6         user.setEmail("email@test.com");
7         user.setPassword("password");
8         user.setName("name");
9         user.setCreatedOn(new Timestamp(new Date().getTime()));
10        userDAO.create(user);
11        Question question = new Question();
12        question.setText("This is a question");
13
14        questionService.create(question, user.getUserId());
15
16        Answer answer = new Answer();
17        answer.setText("This is an answer");
18        question = questionService.addAnswer(answer, question.getQuestionId(),
19                                              user.getUserId());
20
21        assertEquals(1, question.getAnswers().size());
22    }

```

Fixeu-vos en un detall: no heu definit cap objecte `AnswersDAO` i heu pogut persistir una resposta. Recordeu que això és possible gràcies al fet que hem utilitzat

`CascadeType.ALL`, que automàticament s'encarregarà de persistir els objectes que no estiguin associats a una sessió d'Hibernate.

A continuació escriureu el test per a la classe `QuestionService`. En aquest cas el que voldrem fer serà injectar objectes *mock* per a `UserDAO` i `QuestionDAO` i que el test comprovi que la lògica és correcta. La configuració dels objectes *mocks* és a la classe `SpringTestConfig`.

```

1  @Configuration
2  @EnableTransactionManagement
3  @Import(value = {ServiceConfig.class})
4  public class SpringTestConfig {
5      @Bean
6      public UserDAO userDAO() {
7          return Mockito.mock(UserDAO.class);
8      }
9
10     @Bean
11     public QuestionDAO questionDAO() {
12         return Mockito.mock(QuestionDAO.class);
13     }
14
15     @Bean
16     public PlatformTransactionManager transactionManager() {
17         return Mockito.mock(PlatformTransactionManager.class);
18     }
19 }
```

Un cop la configuració dels *beans* està preparada, ja podeu escriure el test.

```

1  import org.ioc.daw.answer.Answer;
2  import org.ioc.daw.config.SpringTestConfig;
3  import org.ioc.daw.user.User;
4  import org.ioc.daw.user.UserDAO;
5  import org.junit.Test;
6  import org.junit.runner.RunWith;
7  import org.mockito.Mockito;
8  import org.springframework.beans.factory.annotation.Autowired;
9  import org.springframework.test.context.ContextConfiguration;
10 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
11
12 import java.util.ArrayList;
13 import java.util.HashSet;
14 import java.util.List;
15 import java.util.Set;
16 import static org.junit.Assert.assertEquals;
17 import static org.junit.Assert.assertNotNull;
18 import static org.junit.Assert.assertSame;
19 import static org.mockito.Mockito.*;
20
21
22 @RunWith(SpringJUnit4ClassRunner.class)
23 @ContextConfiguration(classes = {SpringTestConfig.class})
24 public class QuestionServiceTest {
25     @Autowired
26     private UserDAO userDAOMock;
27
28     @Autowired
29     private QuestionDAO questionDAOMock;
30
31     @Autowired
32     private QuestionService questionService;
33
34     @Test
35     public void getAllQuestions() {
36         Question question1 = getDummyQuestion(1);
37         Set<Question> questions = new HashSet<>();
```

```
38     questions.add(question1);
39
40     int userId = 1;
41     User userMock = Mockito.mock(User.class);
42     when(userMock.getQuestions()).thenReturn(questions);
43
44     when(userDAOMock.getById(userId)).thenReturn(userMock);
45
46     Set<Question> questionsResponse = questionService.getAllQuestions(
47         userId);
48     assertEquals(1, questionsResponse.size());
49 }
50
51 @Test
52 public void addTheFirstAnswerToAQuestion() {
53     int userID = 1;
54     int questionId = 1;
55     int answerId = 1;
56     Answer answer = getDummyAnswer(answerId);
57     Question question1 = getDummyQuestion(questionId);
58     User user1 = getDummyUser(userID);
59
60     when(userDAOMock.getById(userID)).thenReturn(user1);
61     when(questionDAOMock.getById(questionId)).thenReturn(question1);
62     when(questionDAOMock.update(question1)).thenReturn(question1);
63
64     Question questionResponse = questionService.addAnswer(answer,
65         questionId, userID);
66     assertEquals(1, questionResponse.getAnswers().size());
67     verify(userDAOMock, Mockito.times(1)).create(user1);
68 }
69
70 @Test
71 public void addTheAnswerToAQuestionOnceItHasSomeAnswers() {
72     int questionId = 1;
73     int userID = 1;
74
75     Answer answer1 = getDummyAnswer(1);
76     Answer answer2 = getDummyAnswer(2);
77     Question question1 = getDummyQuestion(questionId);
78     Set<Answer> answers = new HashSet<>();
79     answers.add(answer1);
80     question1.setAnswers(answers);
81
82     List<Question> questions = new ArrayList<>();
83     questions.add(question1);
84
85     User user1 = getDummyUser(userID);
86     when(userDAOMock.getById(userID)).thenReturn(user1);
87     when(questionDAOMock.getById(questionId)).thenReturn(question1);
88     when(questionDAOMock.update(question1)).thenReturn(question1);
89
90     questionService.addAnswer(answer2, questionId, userID);
91     verify(userDAOMock, Mockito.times(1)).create(user1);
92     assertEquals(2, answers.size());
93 }
94
95 @Test
96 public void createFirstUserQuestion() {
97     int userId = 1;
98     Question question = getDummyQuestion(1);
99     User user = getDummyUser(1);
100    when(userDAOMock.getById(userId)).thenReturn(user);
101
102    questionService.create(question, userId);
103
104    verify(userDAOMock, timeout(1)).create(user);
105    assertEquals(1, user.getQuestions().size());
106 }
```

```

106
107
108     @Test
109     public void createUserQuestionWhenItsNotTheFirstOne() {
110         int userId = 1;
111         Question question1 = getDummyQuestion(1);
112         Question question2 = getDummyQuestion(2);
113         User user = getDummyUser(1);
114         Set<Question> questions = new HashSet<>();
115         questions.add(question1);
116         user.setQuestions(questions);
117         when(userDAOMock.getById(userId)).thenReturn(user);
118
119         questionService.create(question2, userId);
120
121         verify(userDAOMock, timeout(1)).create(user);
122         assertEquals(2, questions.size());
123     }
124
125     private Question getDummyQuestion(int questionId) {
126         Question question1 = new Question();
127         question1.setQuestionId(questionId);
128         question1.setText("Some question");
129         Set<Question> questions = new HashSet<>();
130         questions.add(question1);
131         return question1;
132     }
133
134     private Answer getDummyAnswer(int answerId) {
135         Answer answer = new Answer();
136         answer.setAnswerId(answerId);
137         answer.setText("This is an answer");
138         return answer;
139     }
140
141     private User getDummyUser(int userId) {
142         String username = "test";
143         User user = new User();
144         user.setUsername(username);
145         user.setUserId(userId);
146         return user;
147     }
148 }
```

Una cosa que heu de comprovar és la diferència de què passa quan creem preguntes o respostes per primer cop. Si és el primer cop, la col·lecció s'ha de crear. Agafem el test `createUserQuestionWhenItsNotTheFirstOne` com a exemple. A la línia 86 afegim una llista de preguntes a l'objecte usuari. Com que la llista existirà, el que farà `QuestionService` serà afegir un element nou, però l'objecte llista serà el mateix que tenia l'usuari. Per això podem comprovar que la llista creada té un objecte més. Al test `createFirstUserQuestion`, l'usuari no té cap pregunta, llavors un nou objecte de tipus llista es crearà i s'associarà a l'objecte `user`, que és el que comprovem a la línia 74.

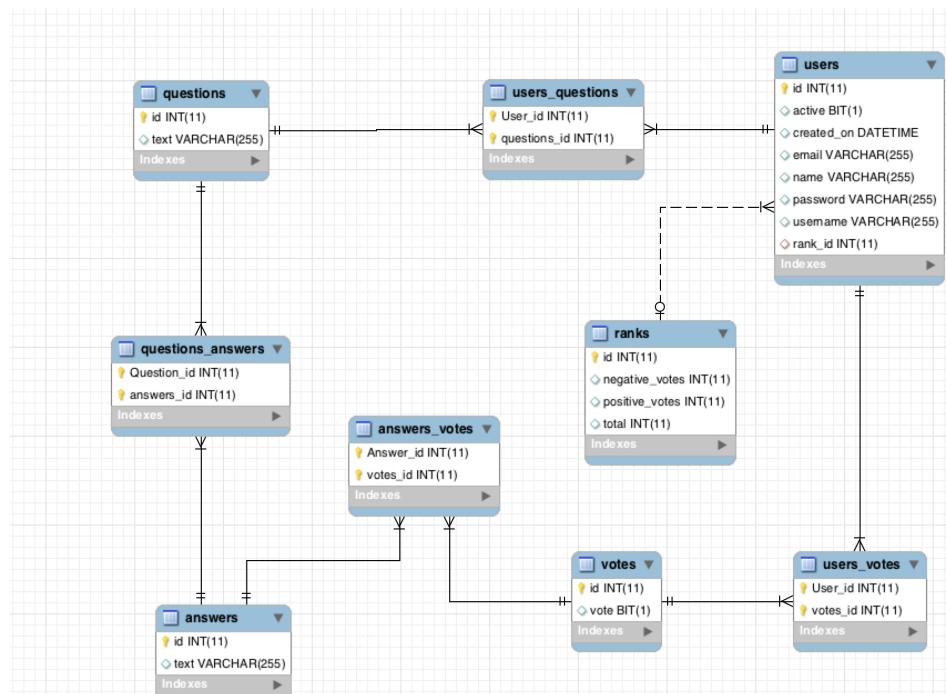
Al fitxer que teniu disponible als annexos de la unitat podeu trobar el codi d'aquest apartat.

3.3 "Socloc". Dialogant amb usuaris, rangs i vots

Un usuari tindrà associat un rang, que serà el resultat d'un càclul dels vots rebuts per les respostes d'un usuari. Els vots poden ser positius o negatius. Així,

un usuari tindrà associada una sèrie de vots, que estaran associats a diferents preguntes. D'aquesta manera, un usuari tindrà múltiples vots i una resposta també pot tenir múltiples vots. Vegeu en la figura 3.12 com serà la relació entre les taules.

FIGURA 3.12. Taules normalitzades



La taula “Votes” està relacionada amb “Answers” i “Users” no directament, però amb unes taules intermèdies . Fixeu-vos també en com la taula “Users” es relaciona amb *rank*, hi ha un camp “user_id” i no hi ha cap taula intermèdia. El motiu és que la relació d'un usuari amb el seu *rank* és 1 a 1, és a dir, cada usuari tindrà només un *rank* i cada *rank* pertany només a un usuari. Vegeu com es representa això en el codi Java de la vostra aplicació. Creeu la classe Rank al paquet org.ioc.daw.rank. Com podeu veure, l'atribut *total* el calcularem a partir dels vots positius i negatius. Cada cop que s'actualitzi el valor dels vots, s'actualitzarà el total.

```

1 import javax.persistence.Column;
2 import javax.persistence.Entity;
3 import javax.persistence.GeneratedValue;
4 import javax.persistence.GenerationType;
5 import javax.persistence.Id;
6 import javax.persistence.Table;
7 import javax.validation.constraints.NotNull;
8 import java.io.Serializable;
9
10 @Entity
11 @Table(name = "ranks")
12 public class Rank implements Serializable {
13     private static final long serialVersionUID = 1L;
14
15     @Id
16     @NotNull
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     @Column(name = "id")
19     private Integer rankId;
20
21     @Column(name = "positive_votes")

```

```

22 private int positive;
23
24     @Column(name = "negative_votes")
25     private int negative;
26
27     @Column(name = "total")
28     private int total;
29
30     public int getPositive() {
31         return positive;
32     }
33
34     public void setPositive(int positive) {
35         this.positive = positive;
36         this.total = positive - getNegative();
37     }
38
39     public int getNegative() {
40         return negative;
41     }
42
43     public void setNegative(int negative) {
44         this.negative = negative;
45         this.total = getPositive() - negative;
46     }
47
48     public Integer getRankId() {
49         return rankId;
50     }
51
52     public void setRankId(Integer rankId) {
53         this.rankId = rankId;
54     }
55
56     public int getTotal() {
57         return total;
58     }
59
60     public void setTotal(int total) {
61         this.total = total;
62     }
63 }
```

A la classe “User” hi afegiu l’atribut rank amb el seu *getter* i *setter*, i també el nou paquet a escanejar per Hibernate a HibernateConfiguration. En la definició de la interfície UserDAO hi ha un mètode relacionat amb el rang. L’implementareu i hi afegireu un test.

```

1 @OneToOne(cascade = {CascadeType.ALL})
2     private Rank rank;
3     public Rank getRank() {
4         return rank;
5     }
6     public void setRank(Rank rank) {
7         this.rank = rank;
8     }
```

```

1 sessionFactory.setPackagesToScan("org.ioc.daw.user", "org.ioc.daw.question",
2                                     "org.ioc.daw.answer", "org.ioc.daw.rank");
```

La implementació que tenim de UserHibernateDAO#findActiveUsers està basada en quan la classe User tenia un atribut de tipus *enter* que tenia el rang de l’usuari. Ara aquesta implementació no funcionarà, ja que l’atribut que conté el rang d’un usuari és una classe, i a la BD, una altra taula. Per obtenir quin és

l'usuari que té un major rang ho podríeu fer mitjançant el llenguatge de consultes d'Hibernate (HQL); el problema és que si canvieu d'implementació de JPA haureu de tornar a escriure totes les consultes de nou. Una solució és utilitzar consultes JPA.

```

1  @Override
2      public User findUserWithHighestRank() {
3          Criteria criteria = createEntityCriteria();
4          criteria.addOrder(Order.desc("rank"));
5          return (User) criteria.uniqueResult();
6      }

```

JPA permet crear consultes amb la classe `CriteriaBuilder`. El que primer indicareu serà de quina classe voldreu fer la consulta (línia 5) i què és el que retornarà la consulta (línia 4) i com s'ordenaran els resultats retornats (línia 7). En aquest cas, l'ordre serà descendent (`cb.desc`) i estarà ordenat per l'atribut total de la classe `Rank`, que és l'atribut `rank` a la classe `User`. Finalment, indiqueu que només voleu retornar el primer resultat (`setMaxResults(1)`) i que per tant aquesta consulta només ha de retornar un únic resultat (`getSingleResult()`).

```

1  @Override
2      public User findUserWithHighestRank() {
3          CriteriaBuilder cb = createCriteriaBuilder();
4          CriteriaQuery<User> criteriaQuery = cb.createQuery(User.class);
5          Root<User> root = criteriaQuery.from(User.class);
6          criteriaQuery.select(root);
7          criteriaQuery.orderBy(cb.desc(root.join("rank").get("total")));
8          EntityManager em = createEntityManager();
9          return em.createQuery(criteriaQuery).setMaxResults(1).getSingleResult()
10         ;
11     }
12     private CriteriaBuilder createCriteriaBuilder() {
13         return getSession().getEntityManagerFactory().getCriteriaBuilder();
14     }
15     private EntityManager createEntityManager() {
16         return getSession().getEntityManagerFactory().createEntityManager();
17     }

```

A continuació creeu la classe `org.ioc.daw.vote.Vote`.

```

1  import javax.persistence.Column;
2  import javax.persistence.Entity;
3  import javax.persistence.GeneratedValue;
4  import javax.persistence.GenerationType;
5  import javax.persistence.Id;
6  import javax.persistence.Table;
7  import javax.validation.constraints.NotNull;
8  import java.io.Serializable;
9
10 @Entity
11 @Table(name = "votes")
12 public class Vote implements Serializable {
13     private static final long serialVersionUID = 1L;
14
15     @Id
16     @NotNull
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     @Column(name = "id")
19     private Integer voteId;
20
21     @Column(name = "vote")
22     private Boolean vote;

```

```

23
24     public Integer getVoteId() {
25         return voteId;
26     }
27
28     public void setVoteId(Integer voteId) {
29         this.voteId = voteId;
30     }
31
32     public Boolean getVote() {
33         return vote;
34     }
35
36     public void setVote(Boolean vote) {
37         this.vote = vote;
38     }
39 }
```

Afegiu el paquet a la configuració d'Hibernate (HibernateConfiguration) per tal que l'escanegi.

```

1 sessionFactory.setPackagesToScan("org.ioc.daw.user", "org.ioc.daw.question",
2                                     "org.ioc.daw.answer", "org.ioc.daw.rank", "org.ioc.daw.vote");
```

Abans de crear el les classes relacionades amb els vots creareu un objecte DAO per a “Answers” i afegireu la relació entre respostes i vots a la classe Answer. Creareu la interfície AnswersDAO i la seva implementació AnswerHibernateDAO, i afegireu el nou DAO a DAOConfig i el *mock* a SpringTestConfig.

```

1 public class Answer implements Serializable {
2     ...
3     @OneToMany(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
4     private Set<Vote> votes;
5
6     public Set<Vote> getVotes() {
7         return votes;
8     }
9
10    public void setVotes(Set<Vote> votes) {
11        this.votes = votes;
12    }
13    ...
14 }
15
16
17 public interface AnswerDAO {
18     Answer getById(Integer questionId);
19     void save(Answer question);
20 }
21
22 @Transactional
23 @Repository("answerHibernateDAO")
24 public class AnswerHibernateDAO implements AnswerDAO {
25     @Autowired
26     private SessionFactory sessionFactory;
27
28     @Override
29     public Answer getById(Integer questionId) {
30         return getSession().get(Answer.class, questionId);
31     }
32
33     @Override
34     public void save(Answer answer) {
35         getSession().saveOrUpdate(answer);
36     }
37 }
```

```

38     protected Session getSession() {
39         return sessionFactory.getCurrentSession();
40     }
41 }
42 }
43
44 @Configuration
45 public class DAOConfig {
46     .....
47     @Bean
48     public AnswerDAO answerDAO(){
49         return new AnswerHibernateDAO();
50     }
51     .....
52 }
```

Un usuari votarà, així que heu d'afegir la relació dels usuaris amb els vots.

```

1 public class User implements Serializable {
2     .....
3     @OneToOne(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
4     private Set<Vote> votes;
5     public Set<Vote> getVotes() {
6         return votes;
7     }
8     public void setVotes(Set<Vote> votes) {
9         this.votes = votes;
10    }
11    .....
```

Com sempre, ara heu de testejar que podem treballar amb l'entitat Votes. Però què és el que realment volem testejar? Realment no volem testejar que podem guardar un vot a la base de dades, ni mai guardarem un vot de forma aïllada. Els vots sempre estaran relacionats amb les respostes i els usuaris. Llavors, el que volem testejar és que un usuari té la capacitat de votar una pregunta i que, si ho fa, aquesta informació es guardarà a la base de dades. Per fer-ho creareu el test VoteDAOTest, encara que no hem creat la classe VoteDAO; el que volem testejar és que els vots es guarden a la base de dades correctament. Abans implementarem la funcionalitat per votar negativament i positivament. Creem la interfície org.ioc.daw.vote.VoteService i la seva implementació. A partir del l'identificador d'una pregunta, recuperem les dades de la BD i creeu el nou objecte de tipus Vote, i en guardar l'objecte pregunta es guardarà la informació del vot.

```

1 public interface VoteService {
2     void votePositive(Integer answerId, Integer userId);
3     void voteNegative(Integer answerId, Integer userId);
4 }
5
6 import org.ioc.daw.answer.Answer;
7 import org.ioc.daw.answer.AnswerDAO;
8 import javax.transaction.Transactional;
9 import java.util.HashSet;
10 import java.util.Set;
11
12 @Transactional
13 public class VoteServiceImpl implements VoteService {
14     private AnswerDAO answerDAO;
15
16     public VoteServiceImpl(AnswerDAO answerDAO, UserDAO userDAO){
17         this.answerDAO = answerDAO;
18         this.userDAO = userDAO;
```

```

19     }
20
21     @Override
22     public void votePositive(Integer answerId, Integer userId) {
23         vote(answerId, userId, true);
24     }
25
26     @Override
27     public void voteNegative(Integer answerId, Integer userId) {
28         vote(answerId, userId, false);
29     }
30
31     private Vote vote(Integer userId, Integer answerId, Boolean value) {
32         User user = userDao.getById(userId);
33         Set<Vote> userVotes = user.getVotes();
34         Vote vote = new Vote();
35         vote.setVote(value);
36         userVotes = getVotes(vote, userVotes);
37         user.setVotes(userVotes);
38         userDao.create(user);
39
40         Answer answer = answerDAO.getById(answerId);
41         Set<Vote> votes = answer.getVotes();
42         votes = getVotes(vote, votes);
43         answer.setVotes(votes);
44         answerDAO.save(answer);
45         return vote;
46     }
47
48     private Set<Vote> getVotes(Vote vote, Set<Vote> votes) {
49         if (votes != null) {
50             votes.add(vote);
51         } else {
52             votes = new HashSet<Vote>();
53             votes.add(vote);
54         }
55         return votes;
56     }
57 }
```

No us heu d'oblidar d'afegir VoteService a ServiceConfig per tal que Spring creï el *bean*.

```

1  @Bean
2  public VoteService voteService(AnswerDAO answerDAO, UserDao userDao){
3      return new VoteServiceImpl(answerDAO, userDao);
4  }
```

Com sempre, ara heu de testejar que podeu treballar amb l'entitat Votes. Però què és el que realment volem testejar? Realment no volem testejar que podem guardar un vot a la base de dades, ni mai guardarem un vot de forma aïllada. Els vots sempre estaran relacionats amb les respostes i els usuaris. Llavors, el que volem testejar és que un usuari té la capacitat de votar una pregunta i que, si ho fa, aquesta informació es guardarà a la base de dades. Per fer-ho creareu el test VoteDAOTest, encara que no hem creat la classe VoteDAO; el que volem testejar és que els vots es guarden a la base de dades correctament.

En el test fareu diverses coses. Primer creeu i persistiu tres usuaris, després l'usuari "user1" crea una pregunta, l'usuari "user2" crea una resposta per a la pregunta i feu que l'usuari "user3" voti de forma positiva la pregunta. Finalment, comproveu que la resposta i l'usuari "user3" tenen un vot i que l'identificador del vot és el mateix en els dos casos.

```
1 import org.ioc.daw.answer.Answer;
2 import org.ioc.daw.answer.AnswerDAO;
3 import org.ioc.daw.config.EmbeddedDatabaseTestConfig;
4 import org.ioc.daw.config.ServicesConfig;
5 import org.ioc.daw.question.Question;
6 import org.ioc.daw.question.QuestionService;
7 import org.ioc.daw.user.User;
8 import org.ioc.daw.user.UserDAO;
9 import org.junit.Test;
10 import org.junit.runner.RunWith;
11 import org.springframework.beans.factory.annotation.Autowired;
12 import org.springframework.test.context.ContextConfiguration;
13 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
14 import java.sql.Timestamp;
15 import java.util.Date;
16 import static org.junit.Assert.assertEquals;
17
18
19 @RunWith(SpringJUnit4ClassRunner.class)
20 @ContextConfiguration(classes = {ServicesConfig.class,
21     EmbeddedDatabaseTestConfig.class})
22 public class VoteDAOCTest {
23     @Autowired
24     private QuestionService questionService;
25
26     @Autowired
27     private UserDAO userDAO;
28
29     @Autowired
30     private AnswerDAO answerDAO;
31
32     @Autowired
33     private VoteService voteService;
34
35     @Test
36     public void votePositive() {
37         User user1 = getUser("test", "test@email.com");
38         User user2 = getUser("test1", "test1@email.com");
39         User user3 = getUser("test2", "test2@email.com");
40         userDAO.create(user1);
41         userDAO.create(user2);
42         userDAO.create(user3);
43
44         Question question = new Question();
45         question.setText("This is a question");
46         questionService.create(question, user1.getUserId());
47
48         Answer answer = new Answer();
49         answer.setText("This is an answer");
50         question = questionService.addAnswer(answer, question.getQuestionId(),
51             user2.getUserId());
52
53         int answerId = question.getAnswers().iterator().next().getAnswerId();
54         voteService.votePositive(user3.getUserId(), answerId);
55
56         User userDB = userDAO.getById(user3.getUserId());
57         Answer answerDB = answerDAO.getById(answerId);
58         assertEquals(1, userDB.getVotes().size());
59         assertEquals(1, answerDB.getVotes().size());
60         assertEquals(userDB.getVotes().iterator().next().getVoteId(), answerDB.
61             getVotes().iterator().next().getVoteId());
62     }
63
64     private User getUser(String username, String email) {
65         User user = new User();
66         user.setUsername(username);
67         user.setActive(true);
```

```
67     user.setEmail(email);
68     user.setPassword("password");
69     user.setName("name");
70     user.setCreatedOn(new Timestamp(new Date().getTime()));
71     return user;
72 }
73 }
```

Podeu trobar el fitxer amb
aquest codi als annexos
de la unitat.

3.4 Què s'ha après?

Heu après que hi ha diferents *frameworks* que ens poden ajudar a l'hora de desenvolupar aplicacions. Spring ens ajuda a crear un codi més modular, reutilitzable i fàcil de testejar. Hem vist com podem canviar fàcilment quina base de dades utilitzar o els *beans* a injectar a una classe. Per una altra banda, Hibernate ens dóna les eines per treballar amb bases de dades focalitzant els esforços en el desenvolupament del codi i no en el disseny de la BD. Això no vol dir que el disseny de la BD no tingui importància; al contrari, serà fonamental per al correcte comportament de l'aplicació quan estigui a producció, però aquesta tasca serà responsabilitat de l'administrador de la BD. Hibernate ens farà més fàcil la tasca de relacionar els objectes de l'aplicació amb les taules de la base dades. També heu après a fer tests unitaris i a testejar l'aplicació utilitzant una BD en memòria emprant els *frameworks* JUnit i Mockito.

Serveis web amb Java EE 7

Josep Maria Camps i Riba

Desenvolupament web en entorn servidor

Índex

Introducció	5
Resultats d'aprenentatge	7
1 Serveis web SOAP amb Java EE 7	9
1.1 Gestió de reserva de places a concerts com a servei web SOAP	11
1.1.1 Creació i configuració inicial del projecte	11
1.1.2 Creació del servei web SOAP	12
1.1.3 Desplegament del servei web SOAP	14
1.1.4 Provant el servei web	17
1.2 Fent servir la gestió de reserva de places a concerts des d'una aplicació Java 'stand-alone'	22
1.3 Fent servir la gestió de reserva de places a concerts des d'una aplicació web	24
1.3.1 Creació i configuració inicial del projecte	24
1.3.2 Creació i prova del client web	25
1.4 Què s'ha après?	28
2 Serveis web RESTful amb Java EE7. Escrivint serveis web	29
2.1 Un servei web RESTful que contesta "Hello World!!!"	30
2.1.1 Creació i configuració inicial del projecte	30
2.1.2 Creació del servei web RESTful	31
2.1.3 Desplegament del servei web RESTful	33
2.2 El servei web de dades de llibres. Operacions CRUD	34
2.2.1 Creació i configuració inicial del projecte	35
2.2.2 Creació i prova del servei web RESTful	37
2.3 Què s'ha après?	43
3 Serveis web RESTful amb Java EE7. Consumint serveis web	45
3.1 Un client per al servei web RESTful que contesta "Hola"	46
3.1.1 Creació i configuració inicial del projecte	46
3.1.2 Creació del client Java 'stand-alone'	46
3.1.3 Desplegament del servei web i prova amb el client Java	47
3.2 El servei web de dades de llibres. Consum i testeig	48
3.2.1 Creació i configuració inicial del projecte	48
3.2.2 Creació i execució dels tests d'integració	50
3.3 Què s'ha après?	57

Introducció

Els **serveis web**, ja siguin **SOAP** o **REST**, són una peça clau en el desenvolupament d'arquitectures de programari orientades a serveis (**SOA**, per les sigles en anglès: Service Oriented Architecture) i, més recentment, per a la creació d'arquitectures basades en microserveis. La seva principal característica és la **interoperativitat**, ja que permeten que aplicacions escrites en llenguatges diferents i que s'executen en plataformes diferents puguin interactuar per construir aplicacions distribuïdes seguint arquitectures SOA.

Veurem els conceptes més rellevants dels serveis web SOAP amb Java EE 7 com a tecnologia. Mitjançant els exemples veureu les bases teòriques que regeixen els serveis web SOAP, quins en són els components més rellevants i quina relació hi ha entre si. Aprendreu a crear-ne, a fer-ne el desplegament i a codificar diferents tipus de clients capaços de consumir els serveis web creats.

Explicarem els conceptes més rellevants dels serveis web RESTful amb Java EE 7 com a tecnologia. Mitjançant els exemples veureu les bases teòriques que regeixen els serveis web RESTful, quins en són els components més rellevants i quina relació hi ha entre si. Aprendreu a crear-ne i a fer-ne el desplegament.

Crearem clients que consumeixin serveis web RESTful amb Java EE 7 com a tecnologia. Mitjançant els exemples, veureu com es codifiquen diferents tipus de client capaços de consumir serveis web RESTful.

La unitat descriu, des d'un vessant teòric i pràctic, els aspectes més essencials dels diferents tipus de serveis web que podem crear. Tots els apartats d'aquesta unitat s'han elaborat seguint exemples pràctics per introduir i aprofundir 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. Desenvolupa serveis web analitzant el seu funcionament i implantant l'estructura dels seus components.

- Identifica les característiques pròpies i l'àmbit d'aplicació dels serveis web.
- Reconeix els avantatges d'utilitzar serveis web per proporcionar accés a funcionalitats incorporades a la lògica de negoci d'una aplicació.
- Determina les tecnologies i els protocols implicats en la publicació i utilització de serveis web.
- Programa un servei web.
- Crea el document de descripció del servei web.
- Verifica el funcionament del servei web.
- Consumeix el servei web.

1. Serveis web SOAP amb Java EE 7

Veurem, mitjançant una aplicació d'exemple, els conceptes més rellevants dels serveis web SOAP. Aprendreu a crear-ne, a fer-ne el desplegament i a codificar diferents tipus de clients capaços de consumir els serveis web creats.

Els serveis web, ja siguin SOAP o REST, són una peça clau en el desenvolupament d'arquitectures de programari orientades a serveis (SOA per les sigles en anglès: Service Oriented Architecture) i, mes recentment, per a la creació d'arquitectures basades en microserveis.

SOAP (de l'anglès *Simple Object Access Protocol*) és un protocol que permet la interacció de serveis web basats en XML. L'especificació de SOAP inclou la sintaxi amb la qual s'han de definir els missatges, les regles de codificació dels tipus de dades i les regles de codificació que regiran les comunicacions entre aquests serveis web.

Quan es descriu una arquitectura **SOAP** ens referim a arquitectures basades en un conjunt de serveis que es despleguen a Internet mitjançant **serveis web**.

Un **servei web** és una tecnologia que fa servir un conjunt de protocols i estàndards per tal d'intercanviar dades entre aplicacions.

Podeu veure els serveis web com a components d'aplicacions distribuïdes que estan disponibles de forma externa i que es poden fer servir per integrar aplicacions escrites en diferents llenguatges (Java, .NET, PHP, etc.) i que s'executen en plataformes diferents (Windows, Linux, etc.). **La seva característica principal és, doncs, la interoperativitat.**

Un servei web publica una lògica de negoci exposada com a servei als clients. La diferència més gran entre una lògica de negoci exposada com a servei web i, per exemple, una lògica de negoci exposada amb un mètode d'un EJB és que els serveis web SOAP proporcionen una interfície poc acoblada als clients. Això permet comunicar aplicacions que s'executin en diferents sistemes operatius, desenvolupades amb diferents tecnologies i amb diferents llenguatges de programació.

Els serveis web i, per tant, les aplicacions orientades a serveis es poden implementar amb diferents tecnologies. Les dues implementacions principals de serveis web que formen part de Java EE 7 són els serveis web **SOAP** i els serveis web **RESTful**.

Els serveis web SOAP són força més complexos que els serveis web RESTful, però proporcionen certes capacitats a nivell de seguretat i transaccionalitat que, a vegades, els fan l'única alternativa viable, sobretot en aplicacions empresarials.

En aquest capítol ens centrem en la creació i el consum de serveis web SOAP amb Java EE 7.

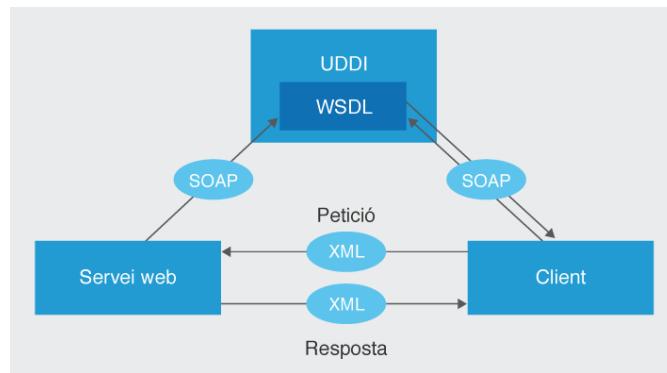
El proveïdor del servei web SOAP el dissenya, l'implementa i en publica la seva interfície i el format dels missatges amb **WSDL** (de l'anglès Web Services Description Language), i els clients consulten aquesta definició i es comuniquen amb el servei web seguint la interfície i el format de missatges que ha publicat el proveïdor del servei en el document WDSL de definició del servei web.

WSDL descriu on es localitza un servei, quines operacions proporciona, el format dels missatges que han de ser intercanviats i com cal cridar el servei.

Opcionalment es pot publicar la definició WSDL del servei web en un registre **UDDI** (de l'anglès Universal Description, Discovery and Integration) per tal que els clients puguin fer-hi cerques. Tingueu en compte que actualment no existeix un registre global que inclogui els diferents serveis web i, per això, UDDI ha passat a ser molt poc utilitzat a la pràctica. Normalment, el client coneix l'adreça del document WSDL de descripció del servei i, a partir d'aquest, invoca el servei web. Vegeu la relació entre els diferents components d'un servei web SOAP en la figura 1.1.

Quan parlem de serveis web, a les operacions se les sol anomenar *endpoints*. Al text farem servir indistintament qualsevol de les dues paraules.

FIGURA 1.1. Components i relacions d'un servei web SOAP



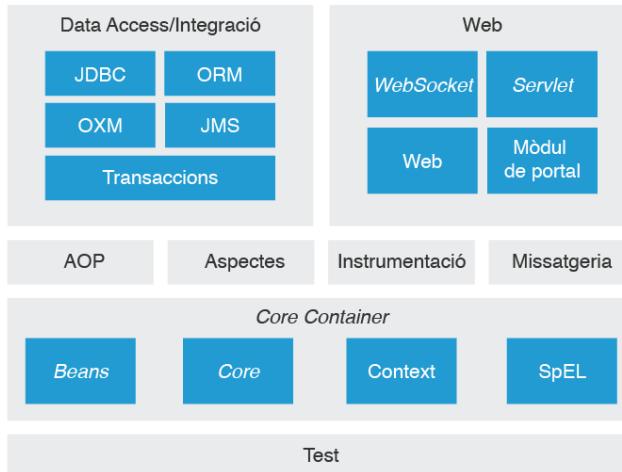
L'especificació de Java EE 7 inclou diverses especificacions que donen complet suport a la creació i el consum de serveis web SOAP; la més destacada és **JAX-WS** (de l'anglès Java API for XML-Based Web Services), que utilitza missatges XML seguint el protocol SOAP i oculta la complexitat d'aquest protocol proporcionant una API senzilla per al desenvolupament, el desplegament i el consum de serveis web SOAP amb Java EE.

JAX-WS és l'API estàndard que defineix Java EE per desenvolupar i desplegar serveis web SOAP i permet ocultar la complexitat inherent a aquest protocol.

Tal com podeu veure en la figura 1.2, el *runtime* de JAX-WS converteix les crides a l'API en missatges SOAP i els missatges SOAP en crides a l'API i ho envia per HTTP. La implementació de JAX-WS que incorporen els servidors d'aplicacions

compatibles amb Java EE 7 també proporciona eines per generar els documents WSDL de definició dels serveis web i eines per generar els artefactes que faran d'intermediaris entre el client i el servei web. Aquests artefactes s'anomenen *stubs* a la part del client i *ties* a la part del servei web.

FIGURA 1.2. Esquema de comunicació entre un servei web SOAP i un client amb JAX-WS



1.1 Gestió de reserva de places a concerts com a servei web SOAP

Veurem els conceptes referents a la definició de serveis web SOAP amb Java EE 7 desenvolupant un servei web SOAP que permeti les següents operacions:

- Proporcionar una llistat de concerts i el nombre de places lliures a cada un d'ells.
- Fer la reserva d'una entrada per a un d'aquests concerts.
- Cancel·lar una reserva d'una entrada.

Farem servir una aplicació web ja desenvolupada que segueixi una arquitectura per capes i hi afegireu una capa de serveis on creareu i publicareu el servei web de gestió de reserves com a servei web SOAP.

Per a aquest exemple farem servir una aproximació *bottom-up*; crearem primer el codi Java que implementarà el servei web i, a partir d'aquest, es generarà automàticament el document WSDL que el descriurà.

Estratègia 'top-down'

Tot i ser força més feixuc, també podríem haver seguit una estratègia *top-down*, creant primer el document de WSDL de definició i, a partit d'aquest, generar automàticament el codi Java que l'implementi.

1.1.1 Creació i configuració inicial del projecte

L'aplicació de la qual partirem s'anomena “Resentioc” i ens servirà per veure com podem exposar algunes funcionalitats de la capa de serveis d'una aplicació

mitjançant serveis web SOAP. Es tracta d'un projecte Spring MVC senzill que segueix una arquitectura típica per capes per tal d'aconseguir una alta reusabilitat, un baix acoblament i una alta cohesió a l'aplicació.

El projecte constarà de quatre capes:

- Capa de presentació
- Capa de domini
- Capa de serveis
- Capa de persistència

A l'exemple no farem servir la capa de presentació, ja que l'objectiu no és proporcionar una interfície d'usuari a l'aplicació, sinó publicar els mètodes de negoci com a serveis web.

El model de domini ja el teniu implementat i és molt senzill, només hi ha una entitat Show que teniu en la figura 1.3 i que representa els concerts que s'estan oferint amb el nombre d'entrades disponibles.

FIGURA 1.3. Entitat Show



La capa de serveis serà la que treballarem i haurà de proporcionar un **servei web SOAP** que permeti als clients fer les següents operacions:

- Llistat de concerts amb el nombre d'entrades disponibles per a cada un d'ells.
- Reservar una entrada.
- Anul·lar una reserva.

La capa de persistència també la teniu implementada i conté l'objecte repositori que permet mapar les dades de la font de dades amb l'objecte de domini. En el nostre cas, per simplificar, farem servir un repositori *in memory* que tindrà una llista precarregada de concerts.

1.1.2 Creació del servei web SOAP

L'objectiu d'aquest apartat és crear un servei web SOAP, seguint l'especificació JAX-WS, amb les tres operacions que corresponen a les tres funcionalitats que volem publicar:

- Llistat de concerts amb el nombre d'entrades disponibles per cada un d'ells.
- Reservar una entrada.
- Anular una reserva.

Els serveis web SOAP amb Java EE 7 es poden implementar de dues maneres: amb una classe Java normal que s'executa en un contingidor de *servlets* o bé com un EJB de sessió sense estat o *singleton* que s'executa en un contingidor d'EJB. Veurem com s'implementa el servei web amb una classe Java normal.

Creeu una interfície que exposarà les tres operacions que volem; anomenarem la interfície *TicketServiceEndpoint* i la crearem al paquet *cat.xtec.ioc.service*.

A la interfície hi creeu els tres mètodes que volem exposar com a *endpoints* del servei web i els anoteu amb l'anotació *javax.jws.WebMethod*:

```

1 package cat.xtec.ioc.service;
2
3 import cat.xtec.ioc.domain.Show;
4 import java.util.ArrayList;
5 import javax.jws.WebMethod;
6 import javax.jws.WebService;
7 import javax.jws.soap.SOAPBinding;
8
9 @WebService
10 @SOAPBinding(style = SOAPBinding.Style.DOCUMENT)
11 public interface TicketServiceEndpoint {
12     @WebMethod ArrayList<Show> getAllShows();
13     @WebMethod Show makeReservation(String showId);
14     @WebMethod Show cancelReservation(String showId);
15 }
```

Fixeu-vos que:

- Hem anotat la interfície amb l'anotació *@WebService* per indicar que la interfície correspondrà a un servei web.
- Hem anotat la interfície amb l'anotació *@SOAPBinding* per tal d'especificar l'estil de servei que volem crear. Per l'exemple, utilitzarem l'estil *Document*, que és l'opció per defecte.
- Tots els mètodes que volem exposar cal que els anoteu amb l'anotació *@WebMethod*.

Creació dels serveis web

Primer ho farem "a mà", sense gaire ajut de NetBeans, per tal que veieu i entengueu tot el procés. Veurem que NetBeans us pot ajudar força en el procés de creació dels serveis web, però és molt important que entengueu tot el procés abans de fer servir eines automatitzades.

Descarregueu el codi del projecte "Resentioc" en l'estat inicial d'aquest apartat des de l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans. Tot i que podeu descarregar-vos també el projecte en l'estat final des de un altre enllaç als annexos, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat.

En les noves versions de Java EE, la creació explícita d'una interfície que caldrà que el servei web implementi és opcional.

Un cop fet això cal que creeu la implementació del servei web; per fer-ho, creeu una nova classe anomenada *TicketServiceEndpointImpl* al paquet *cat.xtec.ioc.service.impl*. En aquesta classe cal que hi implementeu els tres mètodes que volem exposar com a *endpoints* del servei web:

```

1 package cat.xtec.ioc.service.impl;
2
3 import cat.xtec.ioc.domain.Show;
4 import cat.xtec.ioc.domain.repository.ShowRepository;
5 import cat.xtec.ioc.domain.repository.impl.InMemoryShowRepository;
6 import javax.jws.WebService;
```

```

7   import cat.xtec.ioc.service.TicketServiceEndpoint;
8   import java.util.ArrayList;
9
10
11  @WebService(serviceName = "TicketService",
12      endpointInterface = "cat.xtec.ioc.service.TicketServiceEndpoint")
13  public class TicketServiceEndpointImpl implements TicketServiceEndpoint {
14
15      private final ShowRepository showRepository = new InMemoryShowRepository();
16
17      @Override
18      public ArrayList<Show> getAllShows() {
19          return new ArrayList<Show>(this.showRepository.getAllShows());
20      }
21
22      @Override
23      public Show makeReservation(String showId) {
24          return this.showRepository.makeReservation(showId);
25      }
26
27      @Override
28      public Show cancelReservation(String showId) {
29          return this.showRepository.cancelReservation(showId);
30      }
31  }

```

Hem anotat la classe amb l'anotació `@WebService` i li hem indicat amb l'atribut `name` que els clients faran referència al servei web amb el nom `TicketService`. També li hem indicat quina és la interfície que implementa el servei web amb la ruta completa de la interfície `TicketServiceEndpoint`:

```

1  @WebService(serviceName = "TicketService",
2      endpointInterface = "cat.xtec.ioc.service.TicketServiceEndpoint")
3  public class TicketServiceEndpointImpl implements TicketServiceEndpoint {

```

La classe implementa la interfície que defineix el servei web i proporciona una implementació per a cada un dels tres mètodes que implementaran les operacions que publica el servei web cridant mètodes existents del repositori de concerts. A aquests mètodes no cal afegir-hi cap anotació especial; el fet d'implementar una interfície amb mètodes anotats amb `@WebMethod` ja indica quins mètodes de la classe corresponen als *endpoints* del servei web.

I aquesta és tota la feina que ens cal fer per implementar el servei web de reserva d'entrades, ara tan sols ens manca fer-ne el desplegament al servidor d'aplicacions i provar-lo.

1.1.3 Desplegament del servei web SOAP

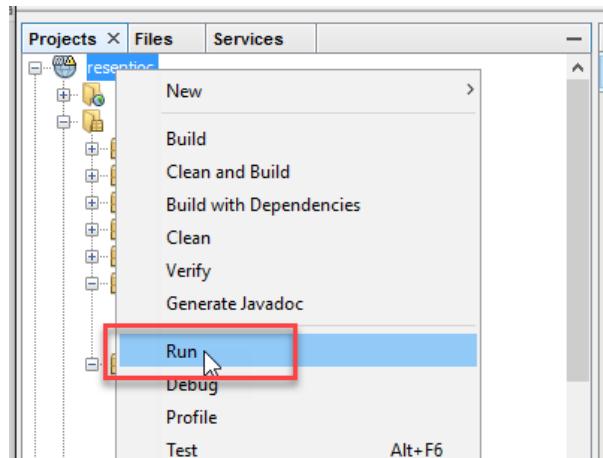
Un cop creat el servei web cal que el desplegueu per tal de fer-lo accessible als clients. El procés de desplegament del servei web engega un conjunt de processos definits a JAX-WS per tal de generar el document WSDL de descripció del servei i publicar els *endpoints* del servei web.

El desplegament del servei web es pot fer de diverses maneres, entre elles:

- Desplegant l'aplicació Java EE que el conté mitjançant els mecanismes normals de desplegament de qualsevol aplicació Java EE.
- Creant una aplicació Java que s'encarregui de publicar el servei web a un URL determinat.

El desplegament del servei web com a part de l'aplicació Java EE que el conté és molt senzill, simplement cal que feu *Run* a NetBeans (vegeu la figura 1.4) i es farà el desplegament al servidor d'aplicacions que tingueu configurat per al projecte:

FIGURA 1.4. 'Run' a NetBeans



En fer el desplegament, la implementació de JAX-WS que té el servidor d'aplicacions ja genera automàticament el document WSDL de definició del servei i publica els *endpoints* que heu definit amb el nom de servei que li heu donat (en el cas que ens ocupa l'hem anomenat *TicketService*). La figura 1.5 mostra la sortida de la consola de NetBeans amb el desplegament del servei web:

FIGURA 1.5. Desplegament a NetBeans

```
Información: Webservice Endpoint deployed TicketServiceEndpointImpl
listening at address at http://BCN-CLI-P113:8080/resentioc/TicketService.
```

Podeu provar que el desplegament des de NetBeans ha anat bé consultant el document WSDL generat en el següent enllaç: localhost:8080/resentioc/TicketService?wsdl.

El desplegament mitjançant una aplicació Java que s'encarregui de publicar el servei web és una mica més complexa, i ens caldrà crear una classe Java que tingui un mètode `main` que cridi el mètode `publish` de l'objecte `javax.xml.ws.Endpoint`. Per fer-ho creem una classe `TicketServicePublisher` al paquet `cat.xtec.ioc.publisher` amb el següent codi:

```
1 package cat.xtec.ioc.publisher;
2
3 import cat.xtec.ioc.service.impl.TicketServiceEndpointImpl;
4 import javax.xml.ws.Endpoint;
5
6 public class TicketServicePublisher {
```

```

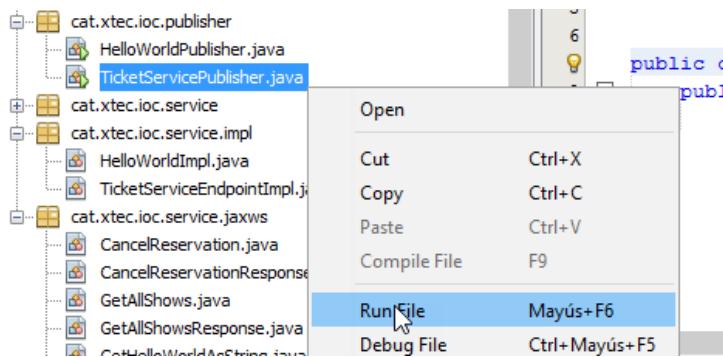
7
8     public static void main(String[] args) {
9         Endpoint.publish("http://localhost:9999/publisher/TicketService", new
10            TicketServiceEndpointImpl());
11    }

```

Fixeu-vos que al mètode publish de l'objecte javax.xml.Endpoint cal que li passeu l'URL on voleu publicar el servei web i una instància de la classe que implementa el servei web.

Amb això ja podeu publicar el servei web de reserva d'entrades executant aquesta aplicació Java; per fer-ho, poseu-vos damunt de la classe a NetBeans i feu *Run File* (vegeu la figura 1.6).

FIGURA 1.6. 'Run File' a NetBeans



Si ho feu, veureu el següent error a la consola de NetBeans:

```

1 Exception in thread "main" com.sun.xml.ws.model.RuntimeModelerException:
2 runtime modeler error: Wrapper class cat.xtec.ioc.service.jaxws.GetAllShows is
   not found. Have you run APT to generate them?

```

Generar artefactes

Quan heu fet el desplegament amb NetBeans és el contenidor del servidor d'aplicacions qui s'encarrega de generar automàticament aquests artefactes, per això no ho heu hagut de fer manualment.

El problema és que ens cal executar una utilitat del JDK anomenada `wsgen` que generarà tots els artefactes portables que necessita el servei web per ser publicat i consumit.

Per fer-ho obriu un *command-prompt*, situeu-vos al directori `\target\classes\` on tingueu el projecte “Resentioc” i executeu la següent línia (cal que tingueu el directori `<JDK_HOME>/bin` al `classpath` per tal que us trobi l'executable `wsgen`):

Amb Windows:

```

1 wsgen -verbose -keep -cp . -s ..\src\main\java cat.xtec.ioc.service.impl.
   TicketServiceEndpointImpl

```

Amb Linux:

```

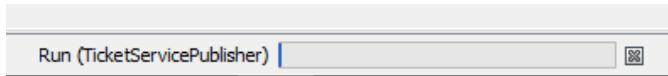
1 wsgen -verbose -keep -cp . -s ../../src/main/java cat.xtec.ioc.service.impl.
   TicketServiceEndpointImpl

```

Aquesta instrucció us generarà un conjunt de fitxers .java amb tots els artefactes portables necessaris per fer la publicació des de l'aplicació Java i els col·locarà al paquet `cat.xtec.ioc.service.jaxws` del projecte “Resentioc”.

Ara sí que podeu publicar el servei web de reserva d'entrades executant el *Publisher*; si tot va bé, veureu que a la part inferior esquerra de NetBeans (vegeu la figura 1.7) us apareix una finestra indicant que el servei web està corrent a l'URL que heu especificat.

FIGURA 1.7. Servei web publicat a NetBeans



Podeu provar que el desplegament des de l'aplicació Java ha anat bé consultant el document WSDL generat en el següent enllaç: localhost:9999/publisher/TicketService?wsdl.

1.1.4 Provant el servei web

Si tot ha anat bé ja teniu el servei web desplegat i a punt per provar-lo. Però com ho fem? Com el provem? Fixeu-vos que fins al moment no hem codificat cap client que faci peticions al servei web desenvolupat; com podem, doncs, provar-lo?

Per tal de provar el servei web que heu desenvolupat, desplegueu l'aplicació al servidor d'aplicacions Glassfish i accediu al següent URL: localhost:8080/resentioc/TicketService?Tester, on veureu una pàgina (vegeu la figura 1.8) amb un botó per a cada una de les operacions que conté el servei web i un enllaç al fitxer WSDL de definició de servei que s'ha generat:

FIGURA 1.8. Pàgina de prova del servei web

TicketService Web Service Tester

This form will allow you to test your web service implementation ([WSDL File](#))

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

Methods :

```
public abstract cat.xtec.ioc.service.Show cat.xtec.ioc.service.impl.TicketServiceEndpoint.cancelReservation(java.lang.String)
cancelReservation (1)
```

```
public abstract java.util.List cat.xtec.ioc.service.impl.TicketServiceEndpoint.getAllShows()
getAllShows ()
```

```
public abstract cat.xtec.ioc.service.Show cat.xtec.ioc.service.impl.TicketServiceEndpoint.makeReservation(java.lang.String)
makeReservation (1)
```

Quan es desplega un servei web SOAP es genera un fitxer WSDL de definició del servei web. El fitxer WSDL està forma per elements XML que **descriuen completament el servei web** i com s'ha de consumir. El podeu consultar en el següent URL: localhost:8080/resentioc/TicketService?WSDL.

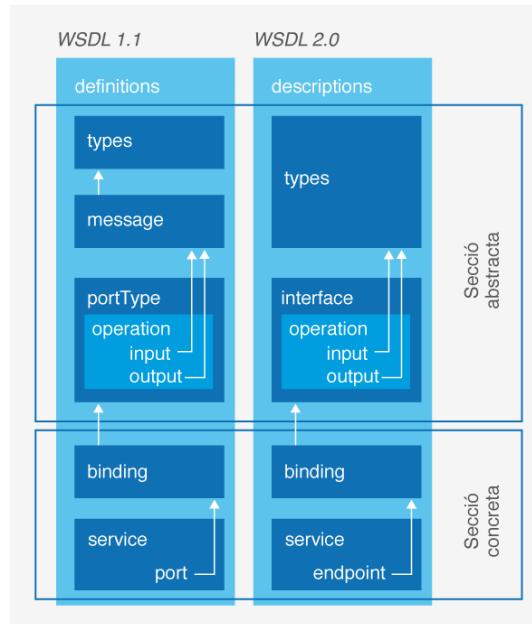
¹ <definitions targetNamespace="http://impl.service.ioc.xtec.cat/" name="TicketService">

```

2   <import namespace="http://service.ioc.xtec.cat/" location="http://
3       localhost:8080/resentioc/TicketService?wsdl=1"/>
4   <binding name="TicketServiceEndpointImplPortBinding" type="
5       ns1:TicketServiceEndpoint"></binding>
6   <service name="TicketService">
7       <port name="TicketServiceEndpointImplPort" binding="
8           tns:TicketServiceEndpointImplPortBinding">
9           <soap:address location="http://localhost:8080/resentioc/
               TicketService"/>
</port>
</service>
</definitions>
```

El fitxer WSDL és un document XML que té una secció abstracta per definir els ports, els missatges i els tipus de dades de les operacions i una part concreta que defineix la instància concreta on hi ha aquestes operacions (vegeu la figura 1.9). Aquesta estructura permet reutilitzar la part abstracta del document.

FIGURA 1.9. Estructura general d'un fitxer WSDL (versions 1.1 i 2.0)



A la part abstracta hi tenim els següents elements:

- types
- message
- portType

Els tipus de dades, tant d'entrada com de sortida, de les operacions es defineixen amb XSD a l'etiqueta <types>. En el nostre cas veiem que ho tenim definit amb un import de localhost:8080/resentioc/TicketService?wsdl=1; si accediu a aquest fitxer veureu que referencia un document d'esquema definit amb XSD:

```

1  <definitions targetNamespace="http://service.ioc.xtec.cat/">
2      <types>
3          <xsd:schema>
```

```

4      <xsd:import namespace="http://service.ioc.xtec.cat/" schemaLocation
5          ="http://localhost:8080/resentioc/TicketService?xsd=1"/>
6  </xsd:schema>
7  </types>
8  <message name="makeReservation"></message>
9  <message name="makeReservationResponse"></message>
10 <message name="getAllShows"></message>
11 <message name="getAllShowsResponse"></message>
12 <message name="cancelReservation"></message>
13 <message name="cancelReservationResponse"></message>
14 <portType name="TicketServiceEndpoint"></portType>
</definitions>
```

I en aquest document XSD localhost:8080/resentioc/TicketService?xsd=1 és on trobem la definició de les operacions i els tipus de dades, tant d'entrada com de sortida, que fan servir les operacions:

```

1 <xs:schema version="1.0" targetNamespace="http://service.ioc.xtec.cat/">
2   <xs:element name="cancelReservation" type="tns:cancelReservation"/>
3   <xs:element name="cancelReservationResponse" type="tns:cancelReservationResponse"/>
4   <xs:element name="getAllShows" type="tns:getAllShows"/>
5   <xs:element name="getAllShowsResponse" type="tns:getAllShowsResponse"/>
6   <xs:element name="makeReservation" type="tns:makeReservation"/>
7   <xs:element name="makeReservationResponse" type="tns:makeReservationResponse"/>
8   <xs:complexType name="cancelReservation">
9     <xs:sequence>
10    <xs:element name="arg0" type="xs:string" minOccurs="0"/>
11  </xs:sequence>
12 </xs:complexType>
13 <xs:complexType name="cancelReservationResponse">
14   <xs:sequence>
15    <xs:element name="return" type="tns:show" minOccurs="0"/>
16  </xs:sequence>
17 </xs:complexType>
18 <xs:complexType name="show">
19   <xs:sequence>
20    <xs:element name="availableTickets" type="xs:int" minOccurs="0"/>
21    <xs:element name="id" type="xs:string" minOccurs="0"/>
22    <xs:element name="location" type="xs:string" minOccurs="0"/>
23    <xs:element name="name" type="xs:string" minOccurs="0"/>
24  </xs:sequence>
25 </xs:complexType>
26 <xs:complexType name="makeReservation">
27   <xs:sequence>
28    <xs:element name="arg0" type="xs:string" minOccurs="0"/>
29  </xs:sequence>
30 </xs:complexType>
31 <xs:complexType name="makeReservationResponse">
32   <xs:sequence>
33    <xs:element name="return" type="tns:show" minOccurs="0"/>
34  </xs:sequence>
35 </xs:complexType>
36 <xs:complexType name="getAllShows">
37   <xs:sequence/>
38 </xs:complexType>
39 <xs:complexType name="getAllShowsResponse">
40   <xs:sequence>
41    <xs:element name="return" type="tns:show" minOccurs="0" maxOccurs="unbounded"/>
42  </xs:sequence>
43 </xs:complexType>
44 </xs:schema>
```

Per exemple, si us hi fixeu, defineix que les peticions a l'operació `cancelReservation` reben un paràmetre de tipus `string` i a les respostes

Aconseguir interoperativitat

El *runtime* de JAX-WS serà l'encarregat de fer les transformacions dels missatges SOAP a crides a l'API Java, fent els mapatges de tipus adient per aconseguir, per exemple, tornar als clients objectes Java complexes de tipus `Show`. Aquesta és la clau per aconseguir interoperativitat, ja que un client que no sigui Java i que tingui un *runtime* que permeti fer les transformacions pertinents també podrà cridar el servei de gestió de reserves que heu creat.

es torna un tipus de dades complex anomenat *show* que està format per un *int* i tres *string* (*availableTickets*, *id*, *location* i *name*)

Després de la definició dels tipus de dades trobem els elements <message> amb la definició dels missatges que es poden intercanviar les operacions del servei web, amb una entrada per a la petició i una altra per a la resposta:

```

1 <definitions targetNamespace="http://service.ioc.xtec.cat/">
2   <types></types>
3   <message name="makeReservation"></message>
4   <message name="makeReservationResponse"></message>
5   <message name="getAllShows"></message>
6   <message name="getAllShowsResponse"></message>
7   <message name="cancelReservation"></message>
8   <message name="cancelReservationResponse"></message>
9   <portType name="TicketServiceEndpoint"></portType>
10 </definitions>
```

I després, els elements <portType> ens defineixen les operacions que es poden fer al servei web i els seus paràmetres:

```

1 <definitions targetNamespace="http://service.ioc.xtec.cat/">
2   <types></types>
3   <message name="makeReservation"></message>
4   <message name="makeReservationResponse"></message>
5   <message name="getAllShows"></message>
6   <message name="getAllShowsResponse"></message>
7   <message name="cancelReservation"></message>
8   <message name="cancelReservationResponse"></message>
9   <portType name="TicketServiceEndpoint">
10    <operation name="makeReservation">
11      <input ns1:Action="http://service.ioc.xtec.cat/
12        TicketServiceEndpoint/makeReservationRequest" message=""
13        tns:makeReservation"/>
14      <output ns2:Action="http://service.ioc.xtec.cat/
15        TicketServiceEndpoint/makeReservationResponse" message=""
16        tns:makeReservationResponse"/>
17    </operation>
18    <operation name="getAllShows">
19      <input ns3:Action="http://service.ioc.xtec.cat/
20        TicketServiceEndpoint/getAllShowsRequest" message=""
21        tns:getAllShows"/>
22      <output ns4:Action="http://service.ioc.xtec.cat/
23        TicketServiceEndpoint/getAllShowsResponse" message=""
24        tns:getAllShowsResponse"/>
25    </operation>
26    <operation name="cancelReservation">
27      <input ns5:Action="http://service.ioc.xtec.cat/
28        TicketServiceEndpoint/cancelReservationRequest" message=""
29        tns:cancelReservation"/>
30      <output ns6:Action="http://service.ioc.xtec.cat/
31        TicketServiceEndpoint/cancelReservationResponse" message=""
32        tns:cancelReservationResponse"/>
33    </operation>
34  </portType>
35 </definitions>
```

Vegem, per exemple, que l'operació *cancelReservation* rep com a paràmetre d'entrada un *tns:cancelReservation* i que retorna un *tns:cancelReservationResponse*. Aquests tipus de dades han estat definides completament en el document XSD (vegeu la figura 1.10).

FIGURA 1.10. Tipus de dades de cancelReservation

```
- <operation name="cancelReservation">
  <input ns5:Action="http://service.ioc.xtec.cat/TicketServiceEndpoint/cancelReservationRequest" message="tns:cancelReservation"/>
  <output ns6:Action="http://service.ioc.xtec.cat/TicketServiceEndpoint/cancelReservationResponse" message="tns:cancelReservationResponse"/>
</operation>
```

A la part concreta hi tenim els següents elements:

- binding
- service

L'element <binding> (vegeu la figura 1.11) defineix el protocol i el format en què es poden fer les operacions; en el nostre cas, es faran per **HTTP** i amb un estil de *Document*.

FIGURA 1.11. Element binding

```
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
```

L'element <service> (vegeu la figura 1.12), per la seva part, defineix el lloc físic on hi ha el servei web (adreça URL) mitjançant una col·lecció de punts de connexió <binding>. En el nostre cas, el servei web es cridarà amb el nom **TicketService** i està desplegat a l'adreça **localhost:8080/resentioc/TicketService**.

FIGURA 1.12. Element service

```
<soap:address location="http://localhost:8080/resentioc/TicketService"/>
```

Si voleu provar alguna de les operacions que proporciona el servei web, per exemple, **getAllShows**, polseu el botó corresponent (vegeu la figura 1.13) i es fa una petició a l'operació del servei web que torna la llista de tots els concerts amb les entrades disponibles.

FIGURA 1.13. Botó per provar l'operació getAllShows

```
<soap:address location="http://localhost:8080/resentioc/TicketService"/>
```

Aquesta petició genera aquest missatge SOAP de petició:

```
1  <?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.
  xsd&nbsp;mlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
  soap/envelope/">
2    <SOAP-ENV:Header/>
3    <S:Body>
4      <ns2:getAllShows xmlns:ns2="http://service.ioc.xtec.cat/">
5    </S:Body>
6  </S:Envelope>
```

I aquest missatge SOAP de resposta:

```
1  <?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.
  xsd&nbsp;mlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
  soap/envelope/">
```

```

2   <SOAP-ENV:Header/>
3   <S:Body>
4       <ns2:getAllShowsResponse xmlns:ns2="http://service.ioc.xtec.cat/">
5           <return>
6               <availableTickets>5</availableTickets>
7               <id>1</id>
8               <location>Palau Sant Jordi</location>
9               <name>U2 360 Tour</name>
10              </return>
11             <return>
12                 <availableTickets>3</availableTickets>
13                 <id>2</id>
14                 <location>Palau de la musica</location>
15                 <name>Carmina Burana</name>
16             </return>
17         </ns2:getAllShowsResponse>
18     </S:Body>
19 </S:Envelope>

```

1.2 Fent servir la gestió de reserva de places a concerts des d'una aplicació Java 'stand-alone'

Hi ha dues maneres de cridar serveis web des de Java amb l'API JAX-WS:

- Creant un *stub* estàtic.
- Fent servir una interfície d'invocació dinàmica.

Anem a crear una aplicació Java *stand-alone* que consumeixi el servei web de gestió de reserves de places a concerts, i ho farem desenvolupant un **client Java stand-alone** que accedirà al servei web amb JAX-WS **utilitzant stubs estàtics**.

Els passos generals per fer un client que faci servir *stubs* estàtics són els següents:

1. Codificar la classe que farà de client.
2. Generar els artefactes necessaris per poder consumir el servei web des d'aquest client amb la utilitat `wsimport`.
3. Compilar i executar el client.

Creeu la classe Java que farà de client al paquet `cat.xtec.ioc.client` i l'anomenieu `TicketServiceClient`.

Genereu els artefactes per tal de poder consumir el servei web amb la utilitat `wsimport`. A `wsimport` cal especificar l'URL del document WSDL de descripció del servei web de gestió de reserves i on voleu que us generi els artefactes. Per fer-ho, obriu un *command-prompt*, situeu-vos al directori `\target\classes\` on tingueu el projecte “Resentclientioc” i executeu la següent línia (cal que tingueu el directori `<JDK_HOME>/bin` al `classpath` per tal que us trobi l'executable `wsimport` i que hagueu fet *Clean and Build* del projecte):

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

Descarregueu el codi del projecte “Resentclientioc” en l'estat inicial d'aquest apartat des de l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

Servidor i WSDL

Assegureu-vos que teniu el servidor d'aplicacions arrencat i el document WSDL de descripció del servei web de gestió de reserves de places a concerts accessible a l'URL que especifiqueu.

Amb Windows:

```
1 wsimport -s ..\src\main\java -p cat.xtec.ioc.service.client.jaxws http://
   localhost:8080/resentioc/TicketService?wsdl
```

Amb Linux:

```
1 wsimport -s ../../src/main/java -p cat.xtec.ioc.service.client.jaxws http://
   localhost:8080/resentioc/TicketService?wsdl
```

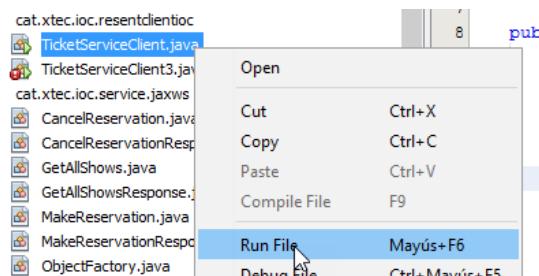
Això us generarà un conjunt de fitxers .java amb tots els artefactes portables necessaris per cridar el servei web des de l'aplicació Java i els col·locarà al paquet *cat.xtec.ioc.service.client.jaxws* del projecte “Resentclientioc”.

Ara ja podem codificar la classe Java *TicketServiceClient* creada utilitzant els artefactes generats per *wsimport*. A les classes Java generades n'hi haurà una que s'anomenarà *TicketService* i que heretarà de *Service*; cal que instancieu aquesta classe i, a partir d'ella, obtingueu l'*stub* que us permetrà cridar les operacions del servei web.

El codi de la classe *TicketServiceClient* amb una crida a l'operació que mostra tots els concerts amb el nombre d'entrades disponibles és el següent:

```
1 package cat.xtec.ioc.client;
2
3 import cat.xtec.ioc.service.client.jaxws.Show;
4 import cat.xtec.ioc.service.client.jaxws.TicketService;
5 import cat.xtec.ioc.service.client.jaxws.TicketServiceEndpoint;
6 import java.util.List;
7
8 public class TicketServiceClient {
9
10    public static void main(String[] args) {
11        TicketService service = new TicketService();
12        TicketServiceEndpoint port = service.getTicketServiceEndpointImplPort()
13            ;
14        List<Show> list = port.getAllShows();
15        printAllShows(list);
16    }
17
18    public static void printAllShows(List<Show> shows) {
19        shows.stream().forEach((show) -> {
20            System.out.println("Show [id=" + show.getId() + ", name=" + show.
21                getName() + ", available tickets=" + show.getAvailableTickets
22                () + "]");
23        });
24    }
25 }
```

Ara ja podem executar el client; per fer-ho, poseu-vos damunt de la classe *TicketServiceClient* i feu *Run File* a NetBeans (vegeu la figura 1.14).

FIGURA 1.14. Execució del client Java a NetBeans

I obtindreu per consola un llistat dels concerts i de les entrades disponibles per a cada un d'ells:

```
1 Show [id=1, name=U2 360 Tour, available tickets=5]
2 Show [id=2, name=Carmina Burana, available tickets=3]
```

1.3 Fent servir la gestió de reserva de places a concerts des d'una aplicació web

Anem a crear una aplicació web que consumeixi el servei web de gestió de reserves de places a concerts.

Per fer-ho crearem una aplicació web molt senzilla que farà servir el servei web SOAP de gestió de reserva de places a concerts.

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

Per desplegar el servei web de gestió de reserva de places a concerts al servidor, descarregueu el codi del projecte resentioc.zip en l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

Per implementar l'aplicació web que accedirà com a client al servei web, descarregueu el codi del projecte resentwebclientioc.zip en l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

1.3.1 Creació i configuració inicial del projecte

Desplegueu el servei web com a part de l'aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans.

El primer que cal fer és desplegar el servei web de gestió de reserva de places a concerts al servidor.

Comproveu que el servei web està desplegat correctament accedint a l'URL localhost:8080/resentioc/TicketService?WSDL amb un navegador.

Un cop desplegat el servei web ens cal un altre projecte, que serà l'aplicació web que accedirà com a client al servei web.

No entrarem en detalls, però el projecte que tenim com a punt de partida és una senzilla aplicació web Spring MVC que ha de mostrar un llistat dels concerts disponibles i, per a cada concert, ha de proporcionar una acció que permeti reservar una entrada i una acció que permeti cancel·lar una reserva. Per simplicitat, a l'exemple no seguirem estrictament una arquitectura per capes on els serveis es criden des de la capa de serveis. Si ho féssim així crearíem un servei propi en

aquesta capa i aquest servei seria l'encarregat de cridar el servei web SOAP de reserva de places a concerts.

L'aplicació web utilitzarà el servei web SOAP de gestió de reserva de places a concerts per proporcionar aquestes funcionalitats.

1.3.2 Creació i prova del client web

Un cop tingueu el projecte “Resentwebclientioc” carregat a NetBeans, el primer que farem serà llistar tots els concerts disponibles. Per fer-ho heu d'accedir a la classe ShowController del paquet cat.xtec.ioc.controller que fa de controlador i crear la referència al servei web SOAP que ens permetrà obtenir la llista de concerts:

```
1 private final TicketService showsWebService=new TicketService();
```

Si ho feu, veureu que NetBeans indica que la classe no compila. Això és degut al fet que ens falta generar els artefactes de client per tal de poder consumir el servei web amb la utilitat wsimport. A wsimport cal especificar l'URL del document WSDL de descripció del servei web de gestió de reserves i on voleu que us generi els artefactes.

Per fer-ho, obriu un *command-prompt*, situeu-vos al directori \target\classes\ on tingueu el projecte “Resentwebclientioc” i executeu la següent línia (cal que tingueu el directori <JDK_HOME>/bin al classpath per tal que us trobi l'executable wsimport i que hagueu fet *Clean and Build* del projecte):

Amb Windows:

```
1 wsimport -s ..\src\main\java -p cat.xtec.ioc.service.client.jaxws http://
localhost:8080/resentioc/TicketService?wsdl
```

Amb Linux:

```
1 wsimport -s ../../src/main/java -p cat.xtec.ioc.service.client.jaxws http://
localhost:8080/resentioc/TicketService?wsdl
```

Això us generarà un conjunt de fitxers .java amb tots els artefactes portables necessaris per cridar el servei web des de l'aplicació i els col·locarà al paquet cat.xtec.ioc.service.client.jaxws del projecte “Resentwebclientioc”. Un cop fet això, i l'import corresponent al controlador, la classe ja us compilàrà.

A l'exemple hem vist que sempre ens cal generar els artefactes de client per poder cridar el servei web SOAP, i hem vist com fer-ho amb la utilitat del JDK wsimport. Si feu servir Maven al projecte també ho podeu fer amb un *plugin* per a JAX-WS que configura un *goal* de Maven anomenat wsimport i que s'executa a la fase generate-sources de generació del codi font.

Ja tenim la referència al servei web SOAP; ara crearem l'acció MVC que torni la llista de concerts. Per fer-ho, creeu un mètode anomenat shows amb el següent codi:

Referències a l'“endpoint” del servei

A l'exemple cream directament una instància de l'*endpoint* del servei (invocació programàtica); també es pot utilitzar l'anotació @WebServiceRef per tal que el contenidor injecti automàticament la instància del proxy (o *stub*) del client.

Objectiu de l'exemple

Tingueu en compte que l'objectiu de l'exemple és veure com podem accedir al servei web SOAP des d'una aplicació web i no la construcció de l'aplicació web en si. Per això hi haurà molts detalls propis de l'anatomia de l'aplicació que elidirem o explicarem amb poc detall.

```

1  @RequestMapping(value = "/shows", method = RequestMethod.GET)
2      public ModelAndView shows(HttpServletRequest request, HttpServletResponse
3          response)
4          throws ServletException, IOException {
5      ModelAndView modelview = new ModelAndView("shows");
6      modelview.getModelMap().addAttribute("shows", showsWebService.
7          getTicketServiceEndpointImplPort().getAllShows());
8      return modelview;
9  }

```

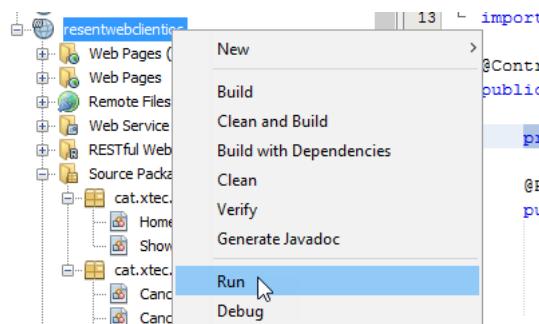
La part important d'aquest mètode és la crida al servei web SOAP:

```
1  showsWebService.getTicketServiceEndpointImplPort().getAllShows();
```

Obtenim una referència al servei web i invoquem el mètode `getAllShows`, que ens tornarà la llista de concerts.

Per provar-ho cal desplegar l'aplicació “Resentwebclientioc” a Glassfish. Per fer-ho, feu *Run* a NetBeans i es farà el desplegament al servidor d'aplicacions que tingueu configurat per al projecte (vegeu la figura 1.15).

FIGURA 1.15. 'Run' a NetBeans



Si tot ha anat bé i accediu a l'URL localhost:8080/resentwebclientioc/shows veureu la pàgina que mostra el llistat de concerts disponibles (vegeu la figura 1.16).

FIGURA 1.16. Llistat de concerts

The screenshot shows a web page titled "Concerts". Below the title is the subtitle "Llista de concerts". There are two main sections displayed:

- U2 360 Tour** at **Palau Sant Jordi**. It states "Hi ha 5 tickets disponibles" and has two buttons: "Reservar una entrada" and "Anul·lar una reserva".
- Carmina Burana** at **Palau de la musica**. It states "Hi ha 3 tickets disponibles" and has two buttons: "Reservar una entrada" and "Anul·lar una reserva".

El procés per crear la funcionalitat que permeti fer una reserva i cancel·lar-la és el mateix: us cal crear l'acció corresponent al controlador i fer ús dels mètodes que proporciona el servei web SOAP de gestió de reserva de places a concerts per implementar-la.

El codi per a aquestes dues accions és el següent:

```

1 @RequestMapping("/makeReservation")
2 public ModelAndView makeReservation(@RequestParam("id") String id,
3     HttpServletRequest request, HttpServletResponse response)
4     throws ServletException, IOException {
5     showsWebService.getTicketServiceEndpointImplPort().makeReservation(id);
6     ModelAndView modelview = new ModelAndView("shows");
7     modelview.getModelMap().addAttribute("shows", showsWebService.
8         getTicketServiceEndpointImplPort().getAllShows());
9     return modelview;
10 }
11
12 @RequestMapping("/cancelReservation")
13 public ModelAndView cancelReservation(@RequestParam("id") String id,
14     HttpServletRequest request, HttpServletResponse response)
15     throws ServletException, IOException {
16     showsWebService.getTicketServiceEndpointImplPort().cancelReservation(id);
17     ModelAndView modelview = new ModelAndView("shows");
18     modelview.getModelMap().addAttribute("shows", showsWebService.
19         getTicketServiceEndpointImplPort().getAllShows());
20     return modelview;
21 }
```

Si desplegueu la nova versió de l'aplicació fent *Run* a NetBeans ja podreu provar les funcionalitat que permeten fer una reserva i cancel·lar-la.

Si ho proveu i aneu fent reserves per al concert del grup **U2** veureu que, quan ja no quedin entrades disponibles, el servidor torna un error HTTP 500 (vegeu la figura 1.17).

FIGURA 1.17. Error 500 - No more tickets available!



Això és degut al fet que des del codi del servei web, quan volem fer una reserva per a un concert que no té localitats disponibles, s'està llençant una excepció:

```

1 public void makeTicketReservation() {
2     if(this.availableTickets > 0) {
3         this.availableTickets--;
4     } else {
5         throw new IllegalArgumentException("No more tickets available!");
6     }
7 }
```

Si mirem el *log* del servidor Glassfish veurem aquesta excepció:

```
1  Grave:  No more tickets available!
2  java.lang.IllegalArgumentException: No more tickets available!
3  at cat.xtec.ioc.domain.Show.makeTicketReservation(Show.java:64)
4  at cat.xtec.ioc.domain.repository.impl.InMemoryShowRepository.makeReservation
   (InMemoryShowRepository.java:33)
5  at cat.xtec.ioc.service.impl.TicketServiceEndpointImpl.makeReservation(
   TicketServiceEndpointImpl.java:24)
```

JAX-WS converteix **automàticament** les excepcions de Java en una SOAPFault en format XML que viatjarà al missatge SOAP de resposta.

SOAPFault és el mecanisme que proporciona SOAP per indicar que el servei web cridat ha tingut algun problema.

1.4 Què s'ha après?

Heu vist les bases per al desenvolupament dels serveis web SOAP amb Java EE 7 i les heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après:

- Les nocions bàsiques dels serveis web SOAP amb Java EE.
- A desenvolupar, desplegar i provar un servei web SOAP amb Java EE.
- A desenvolupar i provar un client Java que consulti un servei web SOAP.
- A desenvolupar i provar una aplicació web que consulti un servei web SOAP.

Per aprofundir en aquests conceptes i veure com us pot ajudar NetBeans en la creació i el consum de serveis web SOAP us recomanem que feu les activitats associades a aquest apartat.

2. Serveis web RESTful amb Java EE7. Escrivint serveis web

Explicarem, mitjançant exemples, els conceptes més rellevants dels serveis web RESTful (de l'anglès REpresentational State Transfer), aprendreu a crear-ne i a fer-ne el desplegament mitjançant exemples.

REST no és un protocol, sinó un conjunt de regles i principis que permeten desenvolupar serveis web fent servir HTTP com a protocol de comunicacions entre el client i el servei web, i es basa a definir **accions sobre recursos** mitjançant l'ús dels mètodes GET, POST, PUT i DELETE inherents d'HTTP.

Per a REST, qualsevol cosa que es pugui identificar amb un URI (de l'anglès Uniform Resource Identifier) es considera un recurs, i, per tant, es pot manipular mitjançant accions (també anomenades verbs) especificades a la capçalera HTTP de les peticions seguint el següent conjunt de regles i principis que regeixen REST:

- POST: crea un recurs nou.
- GET: consulta el recurs i n'obté la representació.
- DELETE: esborra un recurs.
- PUT: modifica un recurs.
- HEAD: obté metainformació del recurs.

REST es basa en l'ús d'estàndards oberts en totes les seves parts; així, fa servir URI per a la localització de recursos, HTTP com a protocol de transport, els verbs HTTP per especificar les accions sobre els recursos i els tipus MIME per a la representació dels recursos (XML, JSON, XHTML, HTML, PDF, GIF, JPG, PNG, etc.).

En una comparació ràpida entre REST i SOAP veiem que REST fa servir gairebé sempre HTTP com a mecanisme de comunicació i XML o JSON per intercanviar dades. Un servei REST no té estat i cada URI representa un recurs sobre el qual s'opera amb verbs HTTP. Un servei web REST no requereix de missatges SOAP/XML ni de descripcions del servei amb documents WSDL de definició de servei.

Als serveis web SOAP tota la infraestructura es basa en XML i les operacions cal que les defineixi el desenvolupador del servei; són força més complexos, però proporcionen certes capacitats a nivell de seguretat i transaccionalitat que, a vegades, els fan l'única alternativa viable, sobretot en aplicacions empresarials. Tot això fa que REST sigui molt més lleuger i fàcil d'utilitzar que SOAP i, per a determinades arquitectures, sigui una millor opció.

Format JSON

JSON (de l'anglès Java Script Object Notation) és un format lleuger d'intercanvi de dades. És facil de llegir i escriure per als éssers humans i, per a les màquines, d'analitzar i generar. Això el fa ideal per representar els recursos en arquitectures REST. Un dels principals problemes dels serveis web basats en SOAP és la mida dels missatges d'intercanvi; l'ús de JSON permet minimitzar la informació a enviar.

Ens centrarem en la creació i el desplegament de serveis web RESTful amb Java EE 7 i ho farem mitjançant l'**API JAX-RS**. Noteu que per escriure un servei web RESTful tan sols us caldrà un client i un servidor que es puguin comunicar per HTTP, però hauríeu de fer a mà tota la configuració, el parseig de les peticions segons el verb HTTP emprat, el mapatge entre el format de dades de la petició i els objectes Java que representen el domini i enviar les respostes amb el tipus MIME que s'especifiqui a la capçalera de la petició; JAX-RS us estalviarà tota aquesta feina proporcionant-vos un petit conjunt d'anotacions que us permetran desenvolupar còmodament serveis web RESTful.

JAX-RS (de l'anglès Java API for RESTful Web Services) és l'API que inclou l'especificació de Java EE 7 per crear i consumir serveis web basats en REST.

2.1 Un servei web RESTful que contesta "Hello World!!!"

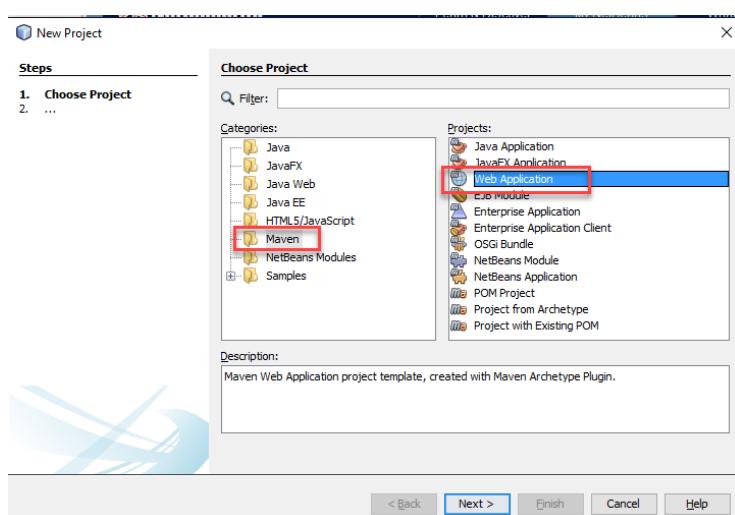
Veurem els diferents conceptes bàsics dels serveis web RESTful amb el típic exemple que sempre trobeu als manuals: farem un "*Hello World!!!*" i el publicarem com a servei web RESTful utilitzant l'API que JAX-RS que proporciona Java EE 7 per fer-ho.

Hem triat per a l'exemple un projecte web amb Maven, però és perfectament vàlid fer-ho amb qualsevol altre tipus de projecte web.

2.1.1 Creació i configuració inicial del projecte

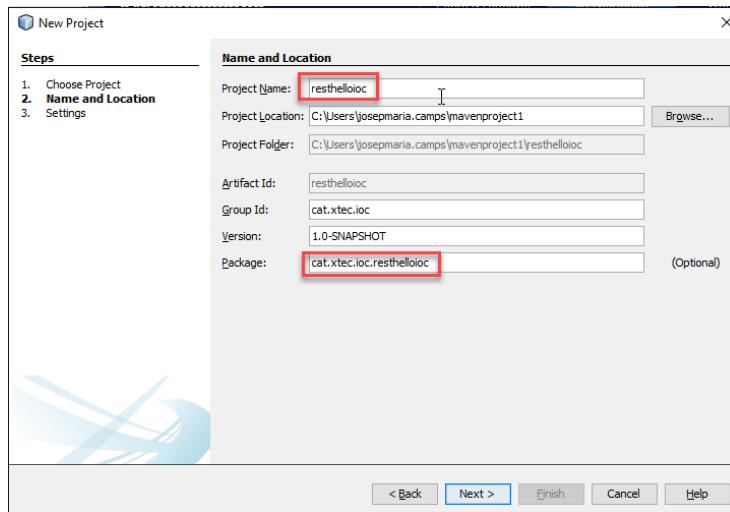
Com que es tracta d'un exemple molt senzill, no ens cal cap projecte de partida; simplement, creeu a NetBeans un nou projecte Maven de tipus *Web Application*. Per fer-ho, feu *File / New Project* i us apareixerà l'assistent de creació de projectes. A l'assistent, seleccioneu *Maven* i *Web Application*, tal com es veu en la figura 2.1.

FIGURA 2.1. Creació de projectes a NetBeans



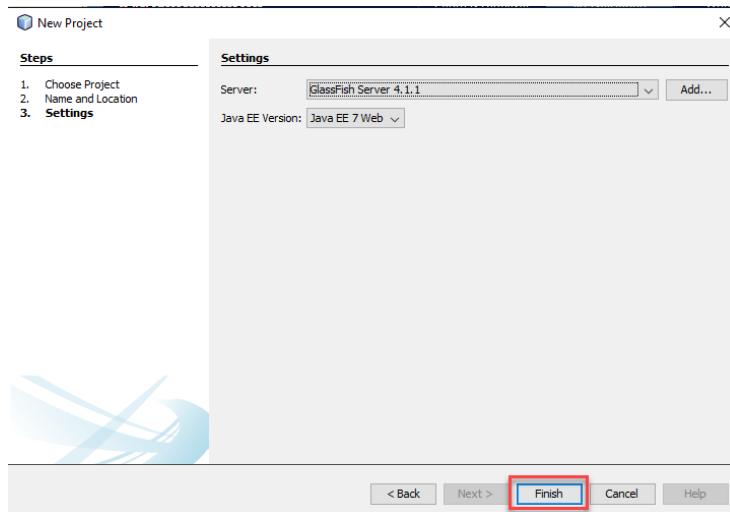
En la següent pantalla (vegeu la figura 2.2) triarem el nom del projecte i el paquet per defecte on anirà el codi font. El podeu anomenar “Resthelloioc” i posar el codi font a `cat.xtec.ioc.resthelloworld`.

FIGURA 2.2. Nom del nou projecte



En la següent pantalla (vegeu la figura 2.3) de l'assistent deixeu els valors per defecte i poleseu *Finish*.

FIGURA 2.3. Configuració del nou projecte



2.1.2 Creació del servei web RESTful

Ja tenim el projecte que ens servirà de base creat. Ara cal que codifiqueu el servei web RESTful que ens ha de tornar la salutació “Hello World!!!”; per fer-ho ens cal decidir primer dues coses: amb quin dels verbs HTTP ha de respondre el servei web i quin URI farem servir per cridar-lo.

Aquest servei web el que ha de fer és **obtenir una representació del recurs** en format HTML; per tant, haurà de respondre al verb GET.

La decisió sobre quin URI utilitzar és força arbitrària en aquest cas; utilitzarem, per exemple, `/hello`.

Per tant, a les peticions d'aquest tipus:

```
1 GET http://localhost:8080/resthelloioc/hello
```

Ha de respondre amb:

```
1 Hello World!!!
```

Un cop decidit el funcionament del nostre servei, el següent que us cal fer és codificar una classe Java que implementi la funcionalitat demandada.

Els serveis web RESTful són **POJO** (de l'anglès Plain Old Java Object) que tenen almenys un mètode anotat amb l'anotació `@Path`.

Creeu una classe Java que implementarà aquesta funcionalitat, anomeneu-la `HelloWorldService` i la creeu al paquet `cat.xtec.ioc.resthelloioc.service`.

Anoteu la classe amb l'anotació `@Path("/hello")` i hi creeu un mètode anomenat, per exemple, `sayHello`, i l'anoteu amb `@GET` i `@Produces("text/html")`.

```
1 package cat.xtec.ioc.resthelloioc.service;
2
3 import javax.ws.rs.GET;
4 import javax.ws.rs.Path;
5 import javax.ws.rs.Produces;
6
7 @Path("/hello")
8 public class HelloWorldService {
9     @GET
10    @Produces("text/html")
11    public String sayHello() {
12        return "Hello World!!!";
13    }
14 }
```

Fixeu-vos que:

- Hem anotat la classe amb `@Path("/hello")` per indicar que respondrà a les peticions que arribin a l'URL localhost:8080/resthelloworld/hello.
- Hem anotat el mètode `sayHello` amb `@GET` per indicar que respondrà a les peticions HTTP que es facin mitjançant el verb GET.
- Hem anotat el mètode `sayHello` amb `@Produces("text/html")` per indicar que respondrà a les peticions HTTP en les quals a la capçalera s'indiqui "text/html" com a tipus MIME, i ho farà produint HTML com a representació del recurs.

I aquesta és tota la feina que ens cal fer per implementar el servei web, JAX-RS farà la resta. Ara tan sols ens manca fer-ne el desplegament al servidor d'aplicacions i provar-lo.

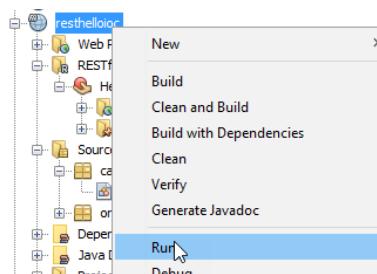
2.1.3 Desplegament del servei web RESTful

Un cop creat el servei web cal que el desplegueu per tal de fer-lo accessible als clients i poder-lo provar.

El procés de desplegament del servei web es fa desplegant l'aplicació Java EE que el conté mitjançant els mecanismes normals de desplegament de qualsevol aplicació Java EE.

El desplegament del servei web com a part de l'aplicació Java EE que el conté és molt senzill: simplement cal que feu *Clean and Build* i després *Run* a NetBeans (vegeu la figura 2.4) i es farà el desplegament al servidor d'aplicacions que tingueu configurat per al projecte.

FIGURA 2.4. 'Run' a NetBeans



Si ho feu i el proveu accedint a l'URL que hem configurat: localhost:8080/resthelloioc/hello veureu que el desplegament falla (vegeu la figura 2.5).

FIGURA 2.5. Error 404 en desplegar



Això és degut al fet que ens cal indicar quines són les classes que ha de tractar com a recursos REST. Això es pot fer de moltes maneres, i una d'elles és crear una classe de configuració anomenada `ApplicationConfig` al paquet `cat.xtec.ioc.resthelloioc.service` amb el següent codi:

```

1 package cat.xtec.ioc.resthelloioc.service;
2
3 import javax.ws.rs.core.Application;
4
5 @javax.ws.rs.ApplicationPath("rest")
6 public class ApplicationConfig extends Application {
7
8 }
```

Si la classe `ApplicationConfig` no es crea al paquet on hi ha les classes que formen els serveis web ens cladrà afegir cada una de les classes a una col·lecció que té la classe `Application` de la qual hereta.

On s'indica que cal afegir “/rest” (o el que vulgueu posar) abans del nom del servei web per cridar-lo.

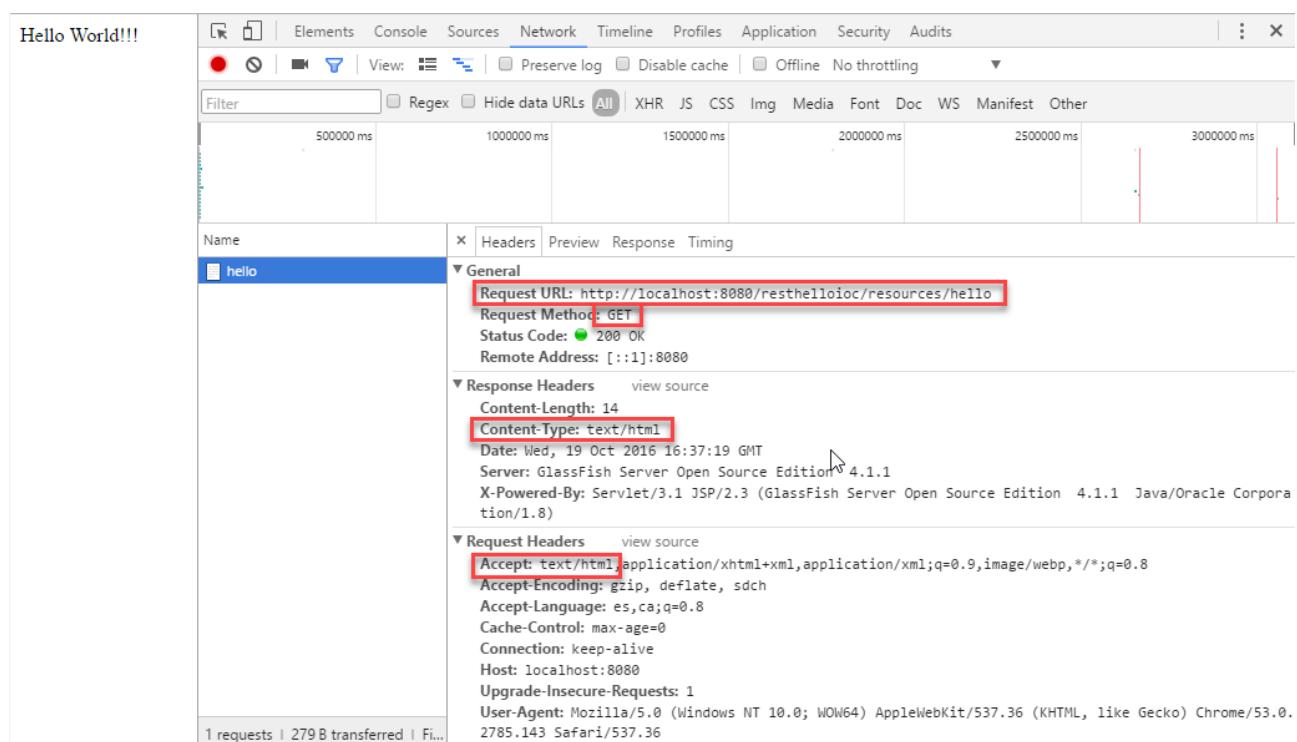
Si proveu ara accedint a l'URL localhost:8080/resthelloioc/rest/hello veureu que el servei web us torna la salutació (vegeu la figura 2.6).

FIGURA 2.6. Servei web desplegat



Si utilitzeu les eines de *debug* de Google Chrome, per exemple, podreu veure la petició HTTP feta i la resposta rebuda (vegeu la figura 2.7).

FIGURA 2.7. Petició i resposta HTTP



2.2 El servei web de dades de llibres. Operacions CRUD

CRUD és l'acrònim en anglès per a les operacions de creació (Create), lectura (Read), actualització (Update) i esborrat (Delete).

Veurem els conceptes referents a la creació de serveis web RESTful amb Java EE 7 desenvolupant un servei web RESTful que permeti treballar sobre un catàleg de llibres. Hi farem les operacions típiques CRUD i dues operacions de cerca: la que ens tornarà tots els llibres i una que ens permetrà fer cerques per títol.

Concretament, el servei web que farem tindrà les següents operacions:

- Consulta d'un llibre mitjançant l'ISBN (operació Read CRUD).
- Creació d'un llibre al catàleg (operació Create CRUD).

- Actualització de les dades d'un llibre (operació Update CRUD).
- Esborrar un llibre del catàleg (operació Delete CRUD).
- Llistar tots els llibres del catàleg.
- Cercar llibres per títol.

Per fer-ho farem servir una aplicació web ja desenvolupada que segueixi una arquitectura per capes i hi afegireu una capa de serveis on creareu i publicareu el servei web de gestió del catàleg com a servei web RESTful. La representació que farem servir per als llibres en tot l'exemple serà JSON.

2.2.1 Creació i configuració inicial del projecte

L'aplicació de la qual partirem s'anomena “Restbooksioc” i ens servirà per veure com podem exposar algunes funcionalitats de la capa de serveis d'una aplicació mitjançant serveis web RESTful. Es tracta d'un projecte Spring MVC senzill que segueix una arquitectura típica per capes per tal d'aconseguir una alta reusabilitat, un baix acoblament i una alta cohesió en l'aplicació.

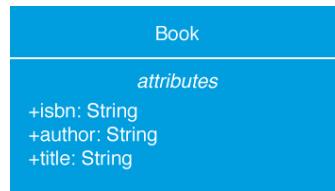
El projecte consta de quatre capes:

- capa de presentació
- capa de domini
- capa de serveis
- capa de persistència

En l'exemple no farem servir la capa de presentació, ja que l'objectiu no és proporcionar una interfície d'usuari a l'aplicació sinó publicar els mètodes de negoci com a serveis web.

El model de domini ja el teniu implementat i és molt senzill, només hi ha una entitat Book que representa els llibres del catàleg (vegeu la figura 2.8).

FIGURA 2.8. Entitat Book



Format JSON

A l'exemple us demanem que treballieu amb una representació JSON. La simplicitat, la lleugeresa i la facilitat de lectura fan ideal aquesta representació per treballar amb aplicacions i dispositius que tenen restriccions pel que fa al volum de dades a intercanviar. Les aplicacions mòbils que consumeixin dades de serveis web RESTful són un molt bon exemple en aquest sentit; la quantitat d'informació que s'intercanviaran client i servidor per fer les operacions és molt menor en una aproximació JSON + RESTful que en una aproximació XML + SOAP.

La representació en JSON d'un recurs llibre és la següent:

```

1  {
2      "isbn": "9788425343537",
3      "author": "Ildefonso Falcones",
4      "title": "La catedral del mar"
5  }

```

Per fer la transformació entre objectes Java i JSON i a l'inrevés cal que importeu Jersey i Jackson com a artefactes al pom.xml. Aquesta tasca ja l'hem fet per vosaltres a la configuració inicial del projecte; les línies afegides al pom.xml són aquestes:

```

1 <dependency>
2     <groupId>org.glassfish.jersey.core</groupId>
3     <artifactId>jersey-server</artifactId>
4     <version>2.22.1</version>
5 </dependency>
6 <dependency>
7     <groupId>com.sun.jersey</groupId>
8     <artifactId>jersey-json</artifactId>
9     <version>1.19</version>
10 </dependency>
11 <dependency>
12     <groupId>org.glassfish.jersey.media</groupId>
13     <artifactId>jersey-media-json-jackson</artifactId>
14     <version>2.22.1</version>
15 </dependency>

```

La capa de serveis serà la que treballarem i haurà de proporcionar **un servei web RESTful** que permeti als clients fer les següents operacions sobre un catàleg de llibres:

- Llistar tots els llibres del catàleg.
- Consulta d'un llibre mitjançant l'ISBN.
- Creació d'un llibre al catàleg.
- Actualització de les dades d'un llibre.
- Esborrar un llibre del catàleg.
- Cercar llibres per títol.

Repository 'in memory'

Tot i que podríem haver optat per fer l'exemple amb un repositori connectat a una font de dades persistent com una base de dades relacional i utilitzar l'API JPA per definir les entitats del projecte, s'ha considerat que això afegeix "soroll" a l'exemple i us faria portar a terme algunes tasques de configuració que no són pròpies de l'objectiu principal. Per aquest motiu, farem servir un repositori *in memory*.

La capa de persistència també la teniu implementada i conté l'objecte repositori que permet mapar les dades de la font de dades amb l'objecte de domini. En el nostre cas, per simplicitat, farem servir un repositori *in memory* que tindrà una llista precarregada amb els llibres del catàleg.

'Bug' a la versió 4.1.1 de Glassfish

Si teniu la versió 4.1.1 de Glassfish sembla que hi ha un error en treballar amb JAX-RS que fa que quan crideu operacions que involucren la transformació entre JSON i les representacions com a objecte Java de les entitats del domini obtingueu la següent excepció:

```

1  java.lang.NoClassDefFoundError:
2  Could not initialize class org.eclipse.persistence.jaxb.
    BeanValidationHelper

```

Per solucionar-ho cal que descarregueu el fitxer org.eclipse.persistence.moxy.jar que trobareu enllaçat als annexos de la unitat, li tragueu l'extensió .zip, substituïu el que teniu amb el mateix nom al Glassfish, a la carpeta <GLASSFISH_HOME>\glassfish\modules, i reinicieu el servidor.

2.2.2 Creació i prova del servei web RESTful

Primer ens cal indicar quines són les classes que ha de tractar com a recursos REST. Això es pot fer de moltes maneres, una d'elles és crear una classe de configuració anomenada ApplicationConfig al paquet cat.xtec.ioc.service amb el següent codi:

```

1 package cat.xtec.ioc.service;
2
3 import javax.ws.rs.core.Application;
4
5 @javax.ws.rs.ApplicationPath("rest")
6 public class ApplicationConfig extends Application {
7
8 }
```

Descarregueu el codi del projecte "Restbooksioc" de l'enllaç que trobareu als annexos de la unitat, i importeu-lo a NetBeans. Tot i que també podeu descarregar-vos el projecte en l'estat final des dels annexos, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat.

On s'indica que cal afegir "/rest" (o el que vulgueu posar) abans del nom del servei web per cridar-lo.

Un cop fet, crearem una classe que exposarà les operacions que volem; anomenarem la classe BooksRestService, la crearem al paquet cat.xtec.ioc.service i l'anotarem amb @Path('/books') i amb @Singleton.

```

1 @Path("/books")
2 @Singleton
3 public class BooksRestService {
4
5 }
```

Quan NetBeans us demani quins imports voleu afegir especifiqueu els del paquet javax.ws.rs.

Fixeu-vos que:

- L'anotació @Path('/books') indica que el recurs estarà accessible a l'URI /books. Accedirem, doncs, als recursos amb l'URL localhost:8080/restbooksioc/rest/books.
- L'anotació @Singleton ens cal, pel fet que estem utilitzant una repositori *in memory* i necessitem que el servei web no s'inicialitzi cada vegada que es fa una petició, per tal de no perdre els canvis que anem fent al catàleg de llibres. Si féssiu servir una font de dades persistent no us caldria aquesta anotació, ja que els canvis persistirien a cada petició.

Un cop creada la classe cal que hi afegiu una referència al repositori del catàleg de llibres amb el següent codi:

```

1 private BookRepository bookRepository = new InMemoryBookRepository();
```

I ja podem començar a codificar el primer servei que volem oferir; per exemple, la consulta de tots els llibres del catàleg. Per fer-ho creeu un mètode anomenat `getAll` amb el següent codi:

```

1  @GET
2  @Produces(MediaType.APPLICATION_JSON)
3  public List<Book> getAll() {
4      return this.bookRepository.getAll();
5 }
```

Fixeu-vos que:

- Hem anotat el mètode amb l'anotació `@GET` per indicar que aquest mètode respondrà a peticions HTTP de tipus GET.
- Hem anotat el mètode amb `@Produces(MediaType.APPLICATION_JSON)` per indicar que el resultat del mètode serà del tipus MIME “*application/json*”, ja que tornarem la llista de llibres en format JSON.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l’URL localhost:8080/restbooksioc/rest/books; si tot ha anat bé veureu la representació JSON del catàleg de llibres al navegador:

```

1  [{"isbn":"9788425343537","author":"Ildefonso Falcones","title":"La catedral del
   mar"}, 
2  {"isbn":"9788467009477","author":"Jose Maria Peridis Perez","title":"La luz y
   el misterio de las catedrales"}]
```

Passem a crear ara el servei de consulta de llibres per ISBN (l’operació Read de CRUD); per fer-ho, creeu un mètode anomenat `find` amb el següent codi:

```

1  @GET
2  @Path("{isbn}")
3  @Produces(MediaType.APPLICATION_JSON)
4  public Book find(@PathParam("isbn") String isbn) {
5      return this.bookRepository.get(isbn);
6 }
```

Hem anotat el mètode amb l’anotació `@GET` per indicar que aquest mètode respondrà a peticions HTTP de tipus GET.

Hem anotat el mètode amb `@Produces(MediaType.APPLICATION_JSON)` per indicar que el resultat del mètode serà del tipus MIME “*application/json*”, ja que tornarem la informació del llibre en format JSON.

Hem anotat el mètode amb l’anotació `@Path("{isbn}")` per indicar que la informació del llibre la tornarem quan ens arribin peticions GET a l’URI `/books/{isbn}`; és a dir, el que posem darrere de `/books` serà l’ISBN del llibre que volem consultar. Les anotacions `@Path` dels mètodes sempre s’afegeixen a l’anotació de l’arrel del recurs que heu definit a la classe.

El paràmetre que rep el mètode s’ha anotat amb l’anotació `@PathParam("isbn")`. Aquesta és la forma que proporciona JAX-RS per extreure els paràmetres d’una petició. En aquest cas estem extraient un paràmetre del *path*, de l’URI, i el passem com a *String* al mètode `find`.

JAX-RS proporciona un ampli conjunt d'anotacions per fer aquesta tasca fàcil, entre elles destaquem `@PathParam`, `@QueryParam`, `@MatrixParam`, `@CookieParam`, `@HeaderParam` i `@FormParam`.

Per exemple, podríem utilitzar `@QueryParam("year")` per extreure un paràmetre que vingués en una petició d'aquesta forma: <localhost:8080/restbooksioc/rest/books?year=2016>.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL <localhost:8080/restbooksioc/rest/books/9788425343537>; si tot ha anat bé, veureu la representació JSON del llibre consultat al navegador:

```
1 [{"isbn": "9788425343537", "author": "Ildefonso Falcones", "title": "La catedral del
  "mar"}]
```

Creem ara el servei que permeti donar d'alta un llibre al catàleg (l'operació `Create` de CRUD); per fer-ho, creeu un mètode anomenat `create` amb el següent codi:

```
1 @POST
2 @Consumes(MediaType.APPLICATION_JSON)
3 public void create(Book book) {
4     this.bookRepository.add(book);
5 }
```

Hem anotat el mètode amb l'anotació `@POST` per indicar que aquest mètode respondrà a peticions HTTP de tipus POST.

Hem anotat el mètode amb `@Consumes(MediaType.APPLICATION_JSON)` per indicar que la informació del llibre que es vol donar d'alta vindrà en format JSON.

Fixeu-vos que tan sols amb aquesta informació JAX-RS és capaç, quan rep una petició POST a l' URI `/books` amb una representació JSON d'un llibre, de crear un objecte de tipus `Book` i passar-lo al mètode `create`. No és fantàstic?

Ara tocaria provar aquesta funcionalitat, però tenim un problema: com podem enviar peticions POST sense fer una aplicació web amb un formulari? Per fer una petició GET tan sols hem de posar l'URL al navegador i ja ho tenim, però per fer peticions POST ens caldrà alguna utilitat extra. La nostra proposta és que feu servir **cURL**, que és una eina molt útil per fer peticions HTTP de diferents tipus i és multiplataforma. Si feu servir Linux possiblement ja la tingueu instal·lada al sistema; en cas que utilitzeu Windows la podeu descarregar en el següent enllaç: <curl.haxx.se/download.html>, o bé aconseguir un .msi que us faciliti la instal·lació.

La sintaxi de cURL per fer peticions és molt senzilla; per exemple, per consultar tots els llibres del catàleg feu un *command prompt*:

```
1 curl localhost:8080/restbooksioc/rest/books
```

Per provar la funcionalitat que permet afegir un llibre al catàleg desplegueu el projecte fent *Run* a NetBeans i feu una petició POST a l'URL <localhost:8080/restbooksioc/rest/books> especificant la informació del llibre amb JSON; això ho podeu fer amb cURL i la següent comanda:

Swagger i Postman

Per documentar i provar API RESTful teniu moltes alternatives, entre elles destaquem Swagger <swagger.io> i Postman <www.getpostman.com>, que és una extensió per al navegador Google Chrome.
</note>

Afegiu el paràmetre `-v` (`verbose`) a les crides a cURL si voleu veure més informació sobre les peticions i respostes que feu.

```

1 curl -H "Content-Type: application/json" -X POST -d "{\"isbn"
  \":\"9788499301518\", \"author\": \"J.K. Rowling\", \"title\": \"Harry Potter
  y la piedra filosofal\"}" http://localhost:8080/restbooksioc/rest/books

```

Fixeu-vos-hi: li diem que farem una petició POST amb el paràmetre -X, li especifiquem el format JSON del llibre amb el paràmetre -d i indiquem que el format és JSON especificant-ho a la capçalera amb el paràmetre -H.

Executeu la comanda anterior i després consulteu el llistat de llibres amb:

```

1 curl localhost:8080/restbooksioc/rest/books

```

Si tot ha anat bé, veureu que el nou llibre s'ha afegit al catàleg de llibres:

```

1 [{"isbn": "9788499301518", "author": "J.K. Rowling", "title": "Harry Potter y la
  piedra filosofal"}, 
2 {"isbn": "9788425343537", "author": "Ildefonso Falcones", "title": "La catedral del
  mar"}, 
3 {"isbn": "9788467009477", "author": "Jose Maria Peridis Perez", "title": "La luz y
  el misterio de las catedrales"}]

```

Creem ara el servei que permeti modificar la informació d'un llibre del catàleg (l'operació Update de CRUD); per fer-ho, creeu un mètode anomenat update amb el següent codi:

```

1 @PUT
2 @Consumes(MediaType.APPLICATION_JSON)
3 public void edit(Book book) {
4     this.bookRepository.update(book);
5 }

```

Les consideracions en aquest cas són les mateixes que hem vist per al cas de la creació de llibres, l'únic que canvia és que farem servir el verb PUT en lloc del verb POST.

Per provar la funcionalitat que permet modificar la informació d'un llibre del catàleg desplegueu el projecte fent *Run* a NetBeans i feu una petició PUT a l'URL localhost:8080/restbooksioc/rest/books especificant la nova informació del llibre amb JSON; això ho podeu fer amb cURL i la següent comanda:

```

1 curl -H "Content-Type: application/json" -X PUT -d "{\"isbn"
  \":\"9788425343537\", \"author\": \" Ildefonso Falcones\", \"title\": \" LA
  CATEDRAL DEL MAR \"}" http://localhost:8080/restbooksioc/rest/books

```

Executeu la comanda anterior i després consulteu la informació del llibre que heu modificat amb:

```

1 curl localhost:8080/restbooksioc/rest/books/9788425343537

```

Si tot ha anat bé, veureu que el títol del llibre ha canviat a majúscules:

```

1 {"isbn": "9788425343537", "author": " Ildefonso Falcones", "title": " LA CATEDRAL
  DEL MAR "}

```

Creem ara el servei que permeti esborrar un llibre del catàleg (l'operació Delete de CRUD); per fer-ho, creeu un mètode anomenat `remove` amb el següent codi:

```

1 @DELETE
2 @Path("{isbn}")
3 public void remove(@PathParam("isbn") String isbn) {
4     this.bookRepository.delete(isbn);
5 }
```

Les consideracions en aquest cas són les mateixes que hem vist per al cas de la consulta de llibres per ISBN, l'únic que canvia és que farem servir el verb DELETE en lloc del verb GET.

Per provar la funcionalitat que permet esborrar un llibre del catàleg desplegueu el projecte fent *Run* a NetBeans i feu una petició DELETE a l'URL `localhost:8080/restbooksioc/rest/books` especificant l'ISBN del llibre que volem esborrar; això ho podeu fer amb cURL i la següent comanda:

```
1 curl -X DELETE http://localhost:8080/restbooksioc/rest/books/9788425343537
```

Executeu la comanda anterior i després consulteu el llistat de llibres amb:

```
1 curl localhost:8080/restbooksioc/rest/books
```

Si tot ha anat bé, veureu que el llibre ja no el teniu al catàleg de llibres:

```
1 [{"isbn":"9788467009477","author":"Jose Maria Peridis Perez","title":"La luz y el misterio de las catedrales"}]
```

Finalment, codificarem una operació que ens permeti cercar llibres per títol; per fer-ho, creeu un mètode anomenat `findByTitle` amb el següent codi:

```

1 @GET
2 @Path("findByTitle/{title}")
3 @Produces(MediaType.APPLICATION_JSON)
4 public List<Book> findByTitle(@PathParam("title") String title) {
5     return this.bookRepository.findByTitle(title);
6 }
```

Les consideracions en aquest cas són les mateixes que hem vist per al cas d'un llibre per ISBN, l'únic que canvia és que l'anotació `@Path` ara comença amb `findByTitle` i després té el paràmetre `{title}` per indicar que la consulta de llibres per títol la tornarem quan ens arribin peticions GET a l'URI `/books/findByTitle/{title}`, és a dir, el que posarem darrere del `/books/findByTitle/` serà el títol que volem cercar.

Si voleu cercar títols de llibre que continguin espais en blanc amb cURL ho haureu de fer posant %20 enllloc de l'espai en blanc.

Per provar la funcionalitat que permet cercar un llibre del catàleg per títol desplegueu el projecte fent *Run* a NetBeans i feu una petició GET a l'URL `localhost:8080/restbooksioc/rest/books/findByTitle` especificant el títol que volem cercar; això ho podeu fer amb cURL i la següent comanda (cercarem tots els llibres que tenen la paraula "catedral" al títol):

```
1 curl http://localhost:8080/restbooksioc/rest/books/findByTitle/catedral
```

Executeu la comanda anterior i, si tot ha anat bé, veureu que us torna els dos llibres del catàleg que tenen la paraula “catedral” al títol:

```

1  [{"isbn":"9788467009477","author":"Jose Maria Peridis Perez","title":"La luz y
   el misterio de las catedrales"},  

2  {"isbn":"9788425343537","author":"Ildefonso Falcones","title":"La catedral del
   mar"}]
```

I amb això ja heu codificat i provat el servei web RESTful que us permet treballar amb el catàleg de llibres.

El codi final de la classe BooksRestService és el següent:

```

1  @Path("/books")
2  @Singleton
3  public class BooksRestService {
4
5      private BookRepository bookRepository = new InMemoryBookRepository();
6
7      @GET
8      @Produces(MediaType.APPLICATION_JSON)
9      public List<Book> getAll() {
10         return this.bookRepository.getAll();
11     }
12
13     @GET
14     @Path("{isbn}")
15     @Produces(MediaType.APPLICATION_JSON)
16     public Book find(@PathParam("isbn") String isbn) {
17         return this.bookRepository.get(isbn);
18     }
19
20     @GET
21     @Path("findByTitle/{title}")
22     @Produces(MediaType.APPLICATION_JSON)
23     public List<Book> findByTitle(@PathParam("title") String title) {
24         return this.bookRepository.findByTitle(title);
25     }
26
27     @POST
28     @Consumes(MediaType.APPLICATION_JSON)
29     public void create(Book book) {
30         this.bookRepository.add(book);
31     }
32
33     @PUT
34     @Consumes(MediaType.APPLICATION_JSON)
35     public void edit(Book book) {
36         this.bookRepository.update(book);
37     }
38
39     @DELETE
40     @Path("{isbn}")
41     public void remove(@PathParam("isbn") String isbn) {
42         this.bookRepository.delete(isbn);
43     }
44 }
```

2.3 Què s'ha après?

En aquest apartat heu vist les bases pel desenvolupament dels serveis web RESTful amb Java EE 7 i les heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après:

- Les nocions bàsiques dels serveis web RESTful amb Java EE.
- A desenvolupar, desplegar i provar un servei web RESTful senzill amb Java EE.
- A desenvolupar, desplegar i provar un servei web RESTful complex amb Java EE que inclou operacions CRUD sobre un recurs.

Per aprofundir en aquests conceptes i veure com us pot ajudar NetBeans en la creació i el consum de serveis web RESTful us recomanem la realització de les activitats associades a aquest apartat.

3. Serveis web RESTful amb Java EE7. Consumint serveis web

Explicarem, mitjançant exemples, com podem consumir serveis web RESTful remots amb Java.

Un cop codificat el servei web RESTful, el següent que ens cal fer és accedir-hi, i per fer-ho l'únic que ens cal és fer les diferents peticions HTTP a l'URI del servidor que tingui els recursos als quals volem accedir.

Es poden provar els serveis RESTful **amb qualsevol eina que permeti fer peticions HTTP**.

La primera eina en la qual pensaríem tots és un navegador web (Microsoft Internet Explorer, Google Chrome, Mozilla Firefox, etc.). El problema és que els navegadors, per defecte, tan sols poden fer peticions GET i peticions POST. Si voleu fer peticions d'un altre tipus (PUT, DELETE, HEAD) us caldrà instal·lar algun *plugin* al navegador que us ho permeti (Postman és un possible *plugin* per a Google Chrome, però n'hi ha molts).

Una altra opció és utilitzar la utilitat **cURL** que us permet fer tot tipus de peticions HTTP per línia de comandes.

Si el que volem és consumir serveis web RESTful des de Java, l'única opció que hi havia abans de JAX-RS 2.0 era fer servir l'API de baix nivell `java.net.HttpURLConnection` o alguna llibreria propietària que us fes la vida una mica més fàcil.

JAX-RS 2.0 proporciona una API client estàndard que permet fer tota mena de peticions HTTP als serveis web RESTful remots de forma fàcil.

L'API client de JAX-RS és molt senzilla d'utilitzar; moltes vegades tan sols us caldrà utilitzar tres classes: `Client`, `WebTarget` i `Response`.

Amb SOAP, l'API JAX-WS forma part del mateix JDK, mentre que amb JAX-RS cal que incloguem JAX-RS al *classpath* del client. JAX-WS generava un conjunt d'artefactes automàticament per poder connectar amb els serveis web, mentre que JAX-RS no en genera cap; tan sols cal tenir el .jar de la implementació de l'API al *classpath*.

3.1 Un client per al servei web RESTful que contesta "Hola"

Veurem com consumir serveis web RESTful des d'una aplicació Java *stand-alone* utilitzant l'API que proporciona JAX-RS.

3.1.1 Creació i configuració inicial del projecte

Descarregueu el codi del projecte "RestHelloIOC" en l'estat inicial d'aquest apartat des de l'enllaç que trobareu als annexos de la unitat, i importeu-lo a NetBeans. Tot i que també podeu descarregar-vos el projecte en l'estat final des del annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat.

Quan NetBeans us demani quins imports voleu afegir especifiqueu els del paquet javax.ws.rs.

El projecte és un senzill servei web RESTful que respon a peticions GET a l'URI localhost:8080/resthelloioc/rest/hello amb la salutació "Hello World!!!".

3.1.2 Creació del client Java 'stand-alone'

Creeu la classe Java que farà de client al paquet cat.xtec.ioc.resthelloioc.client i l'anomeneu HelloWorldClient amb el següent codi:

```

1  public class HelloWorldClient {
2
3      public static void main(String[] args) {
4          Client client = ClientBuilder.newClient();
5          WebTarget target = client.target("http://localhost:8080/resthelloioc/
6              rest/hello");
7          Invocation invocation = target.request(MediaType.TEXT_HTML).buildGet();
8          Response res = invocation.invoke();
9          System.out.println(res.readEntity(String.class));
10     }

```

Fem servir la interfície Client per construir l'objecte WebTarget.

```
1  Client client = ClientBuilder.newClient();
```

L'objecte WebTarget representa l'URI on enviarem les peticions; en el nostre cas, a localhost:8080/resthelloioc/rest/hello.

```
1  WebTarget target = client.target("http://localhost:8080/resthelloioc/rest/hello
2      ");
```

Un cop tenim l'URI on enviarem les peticions ens cal construir la petició HTTP. Ho fem amb la interfície Invocation, que permet, entre moltes altres coses, especificar el tipus MIME de la petició:

```
1  Invocation invocation = target.request(MediaType.TEXT_HTML).buildGet();
```

En aquest punt, quan construïm la petició, podrem especificar paràmetres, especificar el *path*, els objectes i el format d'aquests per a les peticions POST i PUT, etc. Tot això ho podrem fer mitjançant els mètodes de la interfície Invocation.

La creació de l'objecte `Invocation` no executa encara la petició, i per fer-ho cal que executeu el mètode `invoke`:

```
1 Response res = invocation.invoke()
```

Aquesta crida, en el nostre cas, fa una petició GET a un servei web RESTful que hi ha a localhost:8080/resthelloioc/rest/hello i ens torna el resultat en text/HTML.

Noteu que, pel fet que l'API client de JAX-RS és una API *fluent*, podríem escriure tot el codi anterior d'una forma molt més concisa:

```
1 Response response = ClientBuilder.newClient()
2     .target("http://localhost:8080/resthelloioc/rest/hello")
3     .request(MediaType.TEXT_HTML).get();
```

L'objecte `Response` representa la resposta del servei web i conté, a part del “*Hello World!!!*”, informació de la resposta HTTP rebuda del servei web. Amb `Response` podreu verificar els codis HTTP de retorn, accedir a les capçaleres, a les *cookies* i al valor de retorn.

Accedirem al valor retornat pel servei web amb el mètode `readEntity`; a aquest mètode li heu de passar un objecte que permeti fer la transformació entre la representació del recurs que envia el servidor (en el nostre cas, un senzill `String`, però poden ser tipus complexes) i el tipus de dades que volem. JAX-RS s'encarregarà de fer aquestes transformacions.

```
1 res.readEntity(String.class)
```

Ara ja tan sols ens queda desplegar el servei web i executar el client per veure si es comporta com volem.

3.1.3 Desplegament del servei web i prova amb el client Java

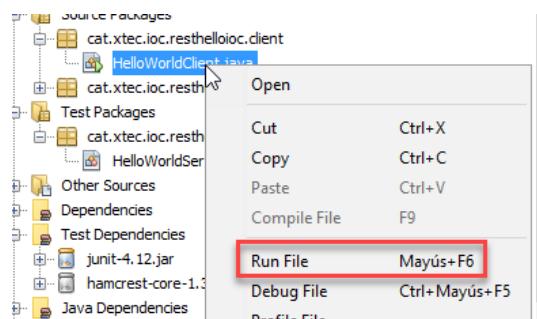
Desplegueu el servei web com a part de l'aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans.

Comproveu que el servei web està desplegat correctament accedint a l'URL localhost:8080/resthelloioc/rest/hello amb un navegador. El servei web us ha de tornar la salutació “*Hello World!!!*”.

Si tot és correcte ja podeu executar el client; per fer-ho, poseu-vos damunt de la classe `HelloWorldClient` i feu *Run File* a NetBeans (vegeu la figura 3.1).

API fluent

Una API fluent és un patró de disseny que permet encadenar les crides als mètodes d'un objecte amb l'objectiu de fer un codi més elegant, concís i comprensible.

FIGURA 3.1. Execució d'una classe Java a NetBeans

I veureu la salutació "*Hello World!!!*" a la consola de sortida de NetBeans:

```

1 Hello World!!!
2
3 BUILD SUCCESS
4

```

3.2 El servei web de dades de llibres. Consum i testeig

Veurem com consumir i fer un test d'integració d'un servei web que permet als clients fer operacions sobre un catàleg de llibres des d'un conjunt de test d'integració amb JUnit i l'API client que proporciona JAX-RS.

El primer que fareu serà desplegar a Glassfish el servei web REST de gestió del catàleg de llibres i després creareu el conjunt de tests d'integració que faran peticions HTTP al servei web amb l'API client de JAX-RS.

Els **tests d'integració** difereixen dels tests unitaris en el fet que no testegen el codi de forma aïllada, sinó que requereixen que el codi a testejar estigui desplegat al servidor d'aplicacions per funcionar.

Descarregueu el codi del projecte "Restbooksioc" des de l'enllaç disponible als annexos de la unitat i importeu-lo a NetBeans.

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'altre enllaç disponible, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat.

Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

3.2.1 Creació i configuració inicial del projecte

Aquest projecte correspon al servei web RESTful de gestió d'un catàleg de llibres i ja té codificades les següents operacions:

- Llistar tots els llibres del catàleg.
- Consulta d'un llibre mitjançant l'ISBN.
- Creació d'un llibre al catàleg.
- Actualització de les dades d'un llibre.

- Esborrar un llibre del catàleg.
- Cercar llibres per títol.

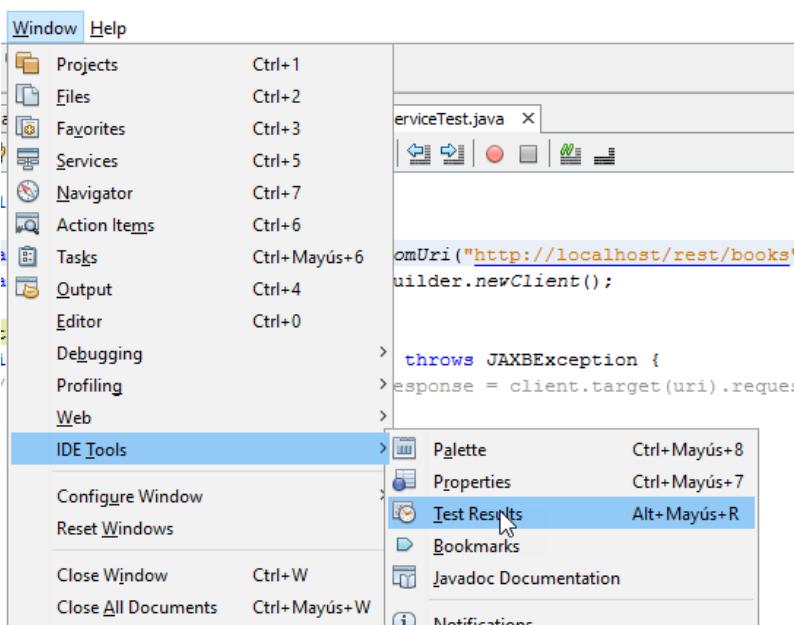
Crearem els tests d'integració dins el mateix projecte que conté el codi del servei web que volem provar. Una altra opció, igualment vàlida, seria crear un nou projecte i posar-hi només el test.

JUnit és un *framework* de test que utilitza anotacions per identificar els mètodes que especificuen un test. A JUnit, un test, ja sigui unitari o d'integració, és un mètode que s'especifica en una classe que només s'utilitza per al test. Això s'anomena *classe de test*. Un mètode de test amb JUnit 4 es defineix amb l'anotació `@org.junit.Test`. En aquest mètode s'utilitza un mètode d'asserció en el qual es comprova el resultat esperat de l'execució de codi en comparació del resultat real.

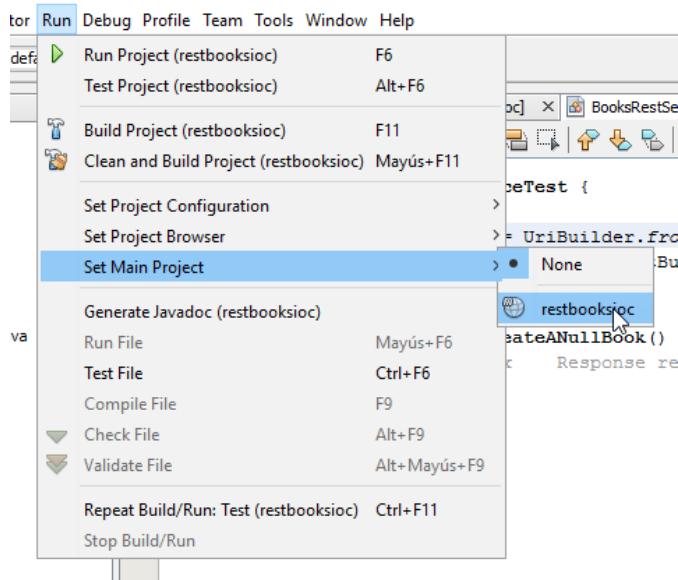
Per tal de fer els tests d'integració farem servir JUnit 4; per fer-ho ens cal tenir la dependència a JUnit al pom.xml del projecte. Al projecte de partida ja hi teniu aquesta dependència afegida.

Tot i que el resultat de l'execució dels tests es pot veure a la finestra de sortida de NetBeans, és molt més còmode visualitzar-ho a la finestra de resultats de test que proporciona també NetBeans. Per mostrar aquesta finestra aneu al menú *Window / IDE Tools / Test Results* (vegeu la figura 3.2).

FIGURA 3.2. Finestra de resultats dels tests



A Netbeans, els tests es poden executar de forma individual, és a dir, classe per classe o tots els del projecte. Si voleu executar tots els tests d'un projecte primer heu de designar com a projecte principal el projecte “Restbooksioc”, tal com podeu veure en la figura 3.3.

FIGURA 3.3. Assignació del projecte com a projecte principal

Un cop fet això, desplegueu el servei web com a part de l’aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans.

Comproveu que el servei web s’ha desplegat correctament accedint a l’URL `localhost:8080/restbooksioc/rest/books` amb un navegador. El servei web us ha de tornar el llistat de llibres que té el catàleg en format JSON:

```

1  [{"isbn":"9788425343537","author":"Ildefonso Falcones","title":"La catedral del
   mar"},  

2  {"isbn":"9788467009477","author":"Jose Maria Peridis Perez","title":"La luz y
   el misterio de las catedrales"}]
```

Quan NetBeans us demani quins imports voleu afegir, especifiqueu els del paquet `javax.ws.rs.`

3.2.2 Creació i execució dels tests d’integració

Ara toca començar a crear els tests d’integració per provar el servei web RESTful de gestió del catàleg.

Per fer-ho, creeu un nou paquet dins de `Test Packages` anomenat, per exemple, `cat.xtec.ioc.test`, i creeu-hi una classe Java anomenada `BooksRestServiceTest` amb el següent codi:

```

1 package cat.xtec.ioc.test;  

2  

3 import javax.ws.rs.client.Client;  

4 import javax.ws.rs.client.ClientBuilder;  

5  

6 public class BooksRestServiceTest {  

7  

8     private static final Client client = ClientBuilder.newClient();  

9  

10 }
```

Estructura dels tests
Un dels patrons més utilitzats a l’hora d’estructurar el codi d’un test és l’anomenat **AAA** (de l’anglès **Arrange-Act-Assert**); amb aquest, els tests sempre tindran una fase de **preparació** (*Arrange*), una d’**execució** (*Act*) i una de **verificació** de resultats (*Assert*).

El primer test que fareu serà un test que **comprovi que la consulta de tots els llibres del catàleg us torna la llista sencera de llibres**; per fer-ho, creeu un mètode

anomenat `shouldReturnAllBooks` dins la classe `BooksRestServiceTest` amb el següent codi:

```

1  @Test
2  public void shouldReturnAllBooks() {
3      // Arrange
4      URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books").
5          port(8080).build();
6      WebTarget target = client.target(uri);
7      Invocation invocation = target.request(MediaType.APPLICATION_JSON).buildGet
8          ();
9      Book first = new Book("9788425343537", "Ildefonso Falcones", "La catedral
10         del mar");
11     Book second = new Book("9788467009477", "Jose Maria Peridis Perez", "La luz
12         y el misterio de las catedrales");
13
14     // Act
15     Response res = invocation.invoke();
16     List<Book> returnedBooks = res.readEntity(new GenericType<List<Book>>() {});
17
18     // Assert
19     assertTrue(returnedBooks.contains(first));
20     assertTrue(returnedBooks.contains(second));
21 }
```

Si analitzeu el codi veureu diverses coses importants: la primera és que heu anotat el mètode `shouldReturnAllBooks` amb l'anotació `@Test` per indicar que es tracta d'un mètode de test.

A la fase de **preparació** feu servir la interfície `Client` per construir l'objecte `WebTarget`, que representa l'URI on enviareu les peticions; en el vostre cas, a localhost:8080/restbooksioc/rest/books.

```

1  URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books").port
2      (8080).build();
3  WebTarget target = client.target(uri);
```

Un cop teniu l'URI on enviareu les peticions cal construir la petició HTTP. Ho feu amb la interfície `Invocation`, que permet, entre moltes altres coses, especificar el tipus MIME de la petició:

```
1  Invocation invocation = target.request(MediaType.APPLICATION_JSON).buildGet();
```

Tot l'anterior ha servit per preparar la petició que voleu enviar al servei web; el següent que fareu serà l'**execució**, cridant el mètode `invoke`:

```
1  Response res = invocation.invoke();
```

Aquesta crida, en el vostre cas, fa una petició GET a un servei web RESTful que hi ha a localhost:8080/restbooksioc/rest/books i torna el resultat en format `application/json`.

L'objecte `Response` representa la resposta del servei web que us permetrà verificar que el resultat és correcte. Per fer-ho, accedireu al valor retornat pel servei web amb el mètode `readEntity`; a aquest mètode li heu de passar un objecte que permeti fer la transformació entre la representació del recurs que envia el servidor

(en el vostre cas, la llista de llibres en format JSON) i el tipus de dades que voleu. JAX-RS s'encarregarà de fer aquestes transformacions.

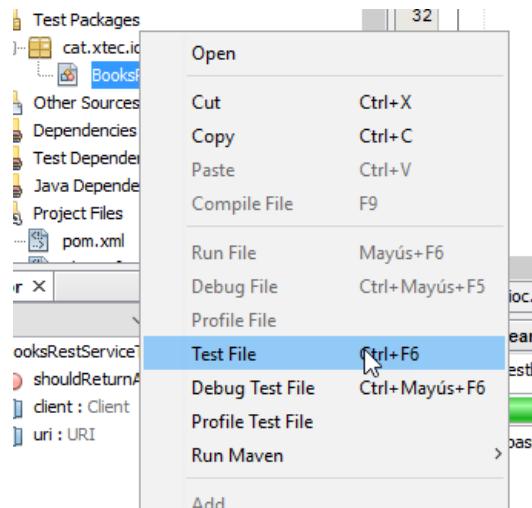
```
1 List<Book> returnedBooks = res.readEntity(new GenericType<List<Book>>() {});
```

En la fase de **verificació** comproveu, per exemple, que el títol dels dos llibres coincideix amb l'esperat:

```
1 assertTrue(returnedBooks.contains(first));
2 assertTrue(returnedBooks.contains(second));
```

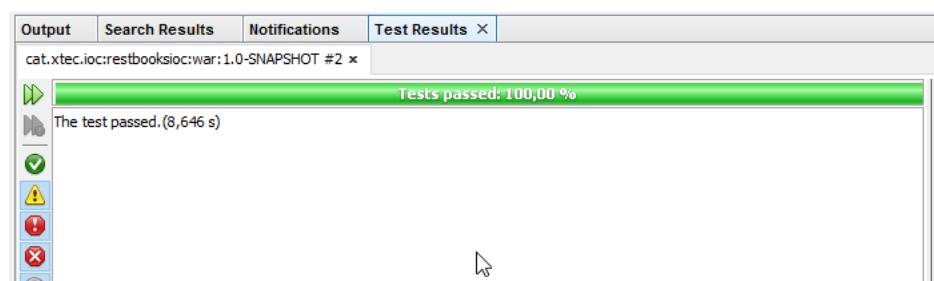
Ja podeu executar el test que heu creat fent *Test File* (vegeu la figura 3.4) al menú contextual de la classe de Test.

FIGURA 3.4. Execució del test



Si tot ha anat bé veureu el resultat a la finestra de Test (vegeu la figura 3.5).

FIGURA 3.5. Resultat de l'execució del test



El segon test que fareu serà un test que **comprovarà que si consulteu un llibre per ISBN que no existeix al catàleg de llibres el servei web us torna el codi HTTP 404 – Not Found.**

Per fer-ho, creeu un mètode anomenat `nonExistentBookShouldReturn404` dins la classe `BooksRestServiceTest` amb el següent codi:

```
1 @Test
2 public void nonExistentBookShouldReturn404() {
3     // Arrange
```

```

4   URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books").
5     port(8080).build();
6   WebTarget target = client.target(uri).path("unknownISBN");
7   Invocation invocation = target.request(MediaType.APPLICATION_JSON).buildGet
8     ();
9
10  // Act
11  Response res = invocation.invoke();
12
13  // Assert
14  assertEquals(Response.Status.NOT_FOUND.toString(), res.getStatusInfo().
15    toString());
16 }
```

Fixeu-vos que ara l'URI on enviareu les peticions GET és `localhost:8080/restbooksioc/rest/books`, i que li afegiu el *path* `/unknownISBN`. Aquesta crida correspon al mètode `find` del servei web que té el següent codi:

```

1 @GET
2 @Path("{isbn}")
3 @Produces(MediaType.APPLICATION_JSON)
4 public Book find(@PathParam("isbn") String isbn) {
5   return this.bookRepository.get(isbn);
6 }
```

A la fase d'execució feu el mateix que per cercar tots els llibres del catàleg, i a la verificació ara comprobareu el codi HTTP retornat per la crida:

```

1 assertEquals(Response.Status.NOT_FOUND.toString(), res.getStatusInfo().toString
2  ());
```

L'objecte `Response` representa la resposta del servei web i conté, a part del contingut, molta informació de la resposta HTTP rebuda. Amb `Response` podeu verificar els codis HTTP de retorn i accedir a les capçaleres, a les *cookies* i al valor de retorn.

Si executeu els tests veureu que el test que comprova que el servei web torni un *404 – Not Found* si consulteu un llibre per ISBN que no existeix falla perquè torna un *204 – No content* enlloc de l'esperat *404 – Not Found* (vegeu la figura 3.6).

FIGURA 3.6. Resultat erroni de l'execució del test



El problema és la codificació del mètode que fa la cerca per ISBN al servei web; us tocarà modificar-lo si voleu que torni un error 404 enlloc d'un 204. Per fer-ho obriu la classe `BooksRestService` del paquet `cat.xtec.ioc.service` i canvieu el codi del mètode `find` pel següent:

```

1 @GET
2 @Path("{isbn}")
3 @Produces(MediaType.APPLICATION_JSON)
4 public Response find(@PathParam("isbn") String isbn) {
5   Book book = this.bookRepository.get(isbn);
```

```

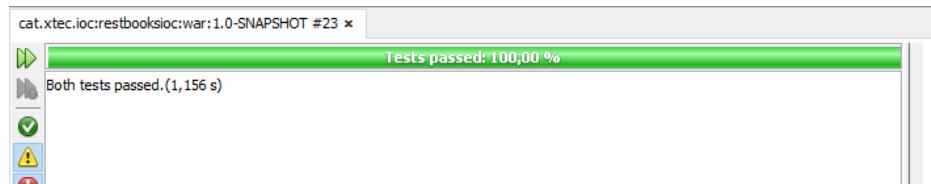
6     if(book == null) {
7         throw new NotFoundException();
8     }
9     return Response.ok(book).build();
10 }

```

Hem canviat el tipus de retorn a `Response`, i en aquest objecte hi afegim el llibre en cas que el trobem. En cas que no trobem el llibre llencem una excepció de tipus `NotFoundException` que el *runtime* de JAX-RS s'encarregarà de transformar en un error *HTTP 404 – Not Found*.

Desplegueu el servei web fent *Run* a NetBeans i torneu a executar el test. Si tot ha anat bé veureu que s'han executat correctament els dos tests (vegeu la figura 3.7).

FIGURA 3.7. Resultat correcte de l'execució dels tests



El tercer test que mostrarem és un test que **comprovarà que no es pugui afegir llibres amb contingut nul**. El que volem és que si la informació del llibre que es vol afegir és nul·la, el servei web ens torni el codi *HTTP 400 – Bad Request*.

Per fer-ho, creeu un mètode anomenat `attemptsToCreateNullBooksShouldReturn400` dins la classe `BooksRestServiceTest` amb el següent codi:

```

1  @Test
2  public void attemptsToCreateNullBooksShouldReturn400() {
3      // Arrange
4      URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books").
5          port(8080).build();
6      WebTarget target = client.target(uri);
7      Invocation invocation = target.request(MediaType.APPLICATION_JSON).
8          buildPost(null);
9
10     // Act
11     Response res = invocation.invoke();
12
13     // Assert
14     assertEquals(Response.Status.BAD_REQUEST.toString(), res.getStatusInfo().
15         toString());
16 }

```

Fixeu-vos que ara enviareu peticions POST a l'URI `localhost:8080/restbooksioc/rest/books` especificant un objecte nul. Aquesta crida correspon al mètode `create` del servei web:

```

1  @POST
2  @Consumes(MediaType.APPLICATION_JSON)
3  public void create(Book book) {
4      this.bookRepository.add(book);
5  }

```

A la fase d'execució feu el mateix que per cercar tots els llibres del catàleg i a la verificació tornareu a comprovar el codi HTTP retornat per la crida:

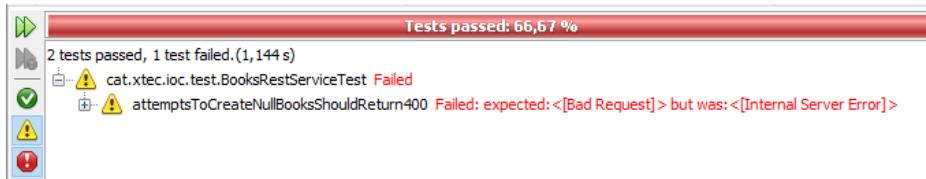
```

1 // Assert
2 assertEquals(Response.Status.BAD_REQUEST.toString(), res.getStatusInfo().
   toString());

```

Si executeu els tests veureu que el test que comprova que no es puguin afegir llibres amb contingut nul falla perquè torna un *500 – Internal Server Error* enllloc del *400 – Bad Request* (vegeu la figura 3.8).

FIGURA 3.8. Resultat erroni de l'execució del test



El problema és la codificació del mètode que fa la creació de llibres al servei web; us tocarà modificar-lo si voleu que torni un error 400 enllloc d'un 500. Per fer-ho, obriu la classe BooksRestService del paquet cat.xtec.ioc.service i canvieu el codi del mètode create pel següent:

```

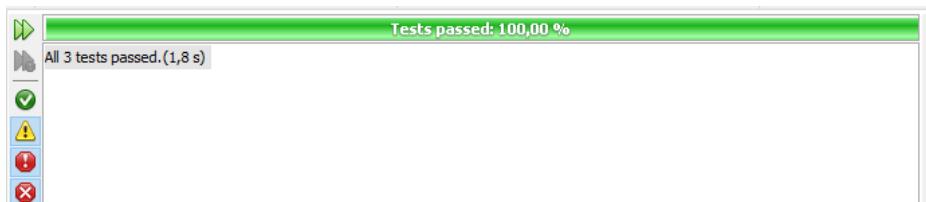
1 @POST
2 @Consumes(MediaType.APPLICATION_JSON)
3 public Response create(Book book) {
4     if(book == null) {
5         throw new BadRequestException();
6     }
7     this.bookRepository.add(book);
8
9     return Response.ok(book).build();
10 }

```

Heu canviat el tipus de retorn a Response, i si us passen un llibre nul llenceu una excepció de tipus BadRequestException que el *runtime* de JAX-RS s'encarregará de transformar en un error *HTTP 400 – Bad Request*.

Desplegueu el servei web fent *Run* a NetBeans i torneu a executar el test. Si tot ha anat bé veureu que s'han executat correctament els tres tests (vegeu la figura 3.9).

FIGURA 3.9. Resultat correcte de l'execució dels tests



El quart i últim test que mostrarem és un test que **comprovarà que la creació d'un llibre ens torni l'URL amb la qual es podrà consultar el llibre**. Per fer-ho, creeu un mètode anomenat createBookShouldReturnTheURLToGetTheBook dins la classe BooksRestServiceTest amb el següent codi:

```

1 @Test
2 public void createBookShouldReturnTheURLToGetTheBook() {

```

```

3   // Arrange
4   Book book = new Book("9788423342518", "Clara Sanchez", "Lo que esconde tu
5     nombre");
6   URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books") .
7     port(8080).build();
8   WebTarget target = client.target(uri);
9   Invocation invocation = target.request(MediaType.APPLICATION_JSON) .
10    buildPost(Entity.entity(book, MediaType.APPLICATION_JSON));
11
12  // Act
13  Response res = invocation.invoke();
14  URI returnedUri = res.readEntity(URI.class);
15
16  // Assert
17  URI expectedUri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/
18    books").port(8080).path("9788423342518").build();
19  assertEquals(expectedUri, returnedUri);
20 }
```

El codi del test és el mateix que el codi del test que comprovava que no es poguessin afegir llibres nul; tan sols canvia la comprovació final i el llibre a afegir. Evidentment, si executeu aquest test fallarà, ja que ara el mètode `create` del servei web està tornant la representació del llibre en format JSON i no l'URL on es pot consultar el llibre.

Per fer que torni l'URL cal que modifiqueu un altre cop el codi del servei web. Per fer-ho, obriu la classe `BooksRestService` del paquet `cat.xtec.ioc.service` i canvieu el codi del mètode `create` pel següent:

```

1  @POST
2  @Consumes(MediaType.APPLICATION_JSON)
3  public Response create(Book book) {
4    if(book == null) {
5      throw new BadRequestException();
6    }
7    this.bookRepository.add(book);
8
9    URI bookUri = uriInfo.getAbsolutePathBuilder().path(book.getIsbn()).build()
10   ;
11  return Response.ok(bookUri).build();
12 }
```

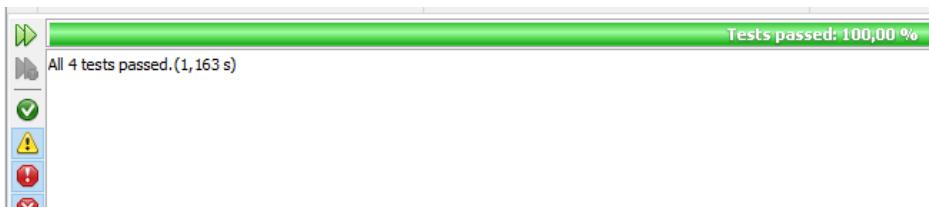
També cal que afegiu el següent atribut a la classe:

```
1  @Context private UriInfo uriInfo;
```

L'objecte `UriInfo` us permet accedir a informació sobre la petició, i amb això podreu construir l'URL amb la qual es podrà consultar el llibre creat i tornar-la a l'objecte `Response`:

```
1  URI bookUri = uriInfo.getAbsolutePathBuilder().path(book.getIsbn()).build();
```

Desplegueu el servei web fent *Run* a NetBeans i torneu a executar el test. Si tot ha anat bé veureu que s'han executat correctament els quatre tests (vegeu la figura 3.10).

FIGURA 3.10. Resultat correcte de l'execució dels tests

Aquests quatre exemples us donen la base per tal que pugueu fer tots els tests d'integració que se us acudeixin i així tenir el servei web de gestió del catàleg de llibres totalment provat!

3.3 Què s'ha après?

En aquest apartat heu vist les bases per al desenvolupament de clients Java que accedeixin a serveis web RESTful amb Java EE 7 i els heu treballat de manera pràctica mitjançant exemples.

Concretament, heu après:

- Les bases de l'API Client de JAX-RS.
- A desenvolupar i provar un client Java *stand-alone* que consulti un servei web RESTful mitjançant l'API Client de JAX-RS.
- A fer tests d'integració que us permeten provar els serveis web RESTful.

Per aprofundir en algun d'aquests conceptes i veure com podeu treballar de forma asíncrona amb els serveis web RESTful us recomanem la realització de l'activitat associada a aquest apartat.

Serveis web amb Spring

Raúl Velaz Mayo

Desenvolupament web en entorn servidor

Índex

Introducció	5
Resultats d'aprenentatge	7
1 Serveis web SOAP amb Spring Web Services	9
1.1 Escrivint un servei web SOAP de consulta de dades d'empreses amb Spring-WS	11
1.1.1 Creació i configuració inicial del projecte	11
1.1.2 Creació del servei web SOAP	12
1.1.3 Desplegament del servei web SOAP	18
1.1.4 Prova del servei web	20
1.2 Fent servir la consulta de dades d'empreses des d'una aplicació Java 'stand-alone'	23
1.3 Què s'ha après?	27
2 Serveis web RESTful amb Spring. Escrivint serveis web	29
2.1 Un servei web RESTful que contesta '"Hello, World!!!"'	30
2.1.1 Creació i configuració inicial del projecte	30
2.1.2 Creació del servei web RESTful	33
2.1.3 Desplegament i prova del servei web RESTful	35
2.2 Testejant el servei web '"Hello, World!!!"'	36
2.2.1 Creació i configuració inicial del projecte	36
2.2.2 Creació i execució dels tests unitaris	37
2.3 El servei web de gestió d'equips de futbol. Operacions CRUD	42
2.3.1 Creació i configuració inicial del projecte	43
2.3.2 Creació i prova del servei web RESTful	44
2.4 Què s'ha après?	52
3 Serveis web RESTful amb Spring. Consumint serveis web	55
3.1 Un client Java per al servei web RESTful '"Hello, World!!!"'	56
3.1.1 Creació i configuració inicial del projecte	56
3.1.2 Creació del client Java 'stand-alone'	57
3.1.3 Desplegament del servei web i prova amb el client Java	58
3.2 Tests d'integració per al servei web RESTful '"Hello, World!!!"'	59
3.2.1 Creació i configuració inicial del projecte	59
3.2.2 Creació i execució dels tests d'integració	60
3.3 Un client JavaScript per al servei web RESTful '"Hello, World!!!"'	63
3.3.1 Creació i configuració inicial del projecte	64
3.3.2 Creació del client JavaScript amb Angular JS	65
3.3.3 Desplegament del servei web i prova del client Angular	66
3.4 Què s'ha après?	67

Introducció

Els **serveis web**, ja siguin **SOAP** o **REST**, són una peça clau en el desenvolupament d'arquitectures de programari orientades a serveis (**SOA**, per les sigles en anglès) i, més recentment, per a la creació d'arquitectures basades en microserveis. La seva principal característica és la **interoperaçivitat**, ja que permeten que aplicacions escrites en llenguatges diferents i que s'executen em plataformes diferents puguin interactuar per construir aplicacions distribuïdes seguit arquitectures SOA.

Spring és un bastiment (*framework* en anglès) amb mòduls basat en Java Enterprise Edition. La principal característica del seu *core* és la utilització del patró de disseny inversió de control (de l'anglès *Inversion of Control*, IoC) i també la injecció de dependències (de l'anglès *Dependency Injection*), un tipus d'IoC. Spring ofereix també un conjunt de projectes/mòduls per desenvolupar serveis web SOAP i REST.

La unitat aborda els conceptes més rellevants dels serveis web SOAP amb Spring-WS com a tecnologia. Mitjançant els exemples, es veuran les bases teòriques que regeixen els serveis web SOAP i com es traslladen i apliquen en la construcció de serveis web SOAP amb Spring-WS, quins en són els components més rellevants i quina relació hi ha entre si. Aprendreu a crear-ne, a fer-ne el desplegament i a codificar clients Java capaços de consumir-los.

Així mateix, s'introduiran els conceptes més rellevants dels serveis web RESTful amb Spring MVC com a tecnologia. Mitjançant els exemples, coneixerem les bases teòriques que regeixen els serveis web RESTful, quins en són els components més rellevants i quina relació hi ha entre si. Aprendreu a crear-ne, a fer-ne el desplegament i a provar-los.

S'explicarà com codificar tests d'integració, clients Java amb Spring i clients AJAX capaços de consumir serveis web RESTful. Mitjançant els exemples, es veurà com es codifiquen aquests tipus de client.

Es descriuran, des d'un vessant teòric i pràctic, els aspectes més essencials dels diferents tipus de serveis web que podem crear amb Spring. Tots els apartats d'aquesta unitat s'han elaborat seguit exemples pràctics per introduir i aprofundir 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. Desenvolupa serveis web analitzant el seu funcionament i implantant l'estructura dels seus components.

- Identifica les característiques pròpies i l'àmbit d'aplicació dels serveis web.
- Reconeix els avantatges d'utilitzar serveis web per proporcionar accés a funcionalitats incorporades a la lògica de negoci d'una aplicació.
- Identifica les tecnologies i els protocols implicats en la publicació i la utilització de serveis web.
- Programa un servei web.
- Crea el document de descripció del servei web.
- Verifica el funcionament del servei web.
- Consumeix el servei web.

2. Genera pàgines web dinàmiques analitzant i utilitzant tecnologies del servidor web que afegeixin codi al llenguatge de marques.

- Identifica les diferències entre l'execució de codi al servidor i al client web.
- Reconeix els avantatges d'unir les dues tecnologies en el procés de desenvolupament de programes.
- Identifica les llibreries i les tecnologies relacionades amb la generació per part del servidor de pàgines web amb guions embedguts.
- Utilitza aquestes tecnologies per generar pàgines web que incloguin interacció amb l'usuari en forma d'advertències i peticions de confirmació.
- Fa servir aquestes tecnologies per generar pàgines web que incloguin verificació de formularis.
- Empra aquestes tecnologies per generar pàgines web que incloguin modificació dinàmica del seu contingut i la seva estructura.
- Aplica aquestes tecnologies en la programació d'aplicacions web.

3. Desenvolupa aplicacions web híbrids seleccionant i utilitzant llibreries de codi i dipòsits heterogenis d'informació.

- Reconeix els avantatges que proporciona la reutilització de codi i l'aprofitament d'informació ja existent.
- Identifica llibreries de codi i tecnologies aplicables en la creació d'applicacions web híbrides.
- Crea una aplicació web que recuperi i processi dipòsits d'informació ja existents.
- Crea dipòsits específics a partir d'informació existent a Internet i en magatzems d'informació.
- Utilitza llibreries de codi per incorporar funcionalitats específiques a una aplicació web.
- Programa serveis i aplicacions web utilitzant com a base informació i codi generats per tercers.
- Prova, depura i documenta les aplicacions generades.

1. Serveis web SOAP amb Spring Web Services

Mitjançant una aplicació d'exemple, veurem els conceptes més rellevants dels serveis web SOAP desenvolupats amb Spring Web Services (Spring-WS). Aprendreu a crear-ne, a fer-ne el desplegament i a codificar diferents tipus de clients capaços de consumir els serveis web creats.

Els serveis web, ja siguin SOAP o REST, són una peça clau en el desenvolupament d'arquitectures de programari orientades a serveis (SOA, per les sigles en anglès) i, més recentment, per a la creació d'arquitectures basades en microserveis.

Quan es descriu una arquitectura SOA ens referim a arquitectures basades en un conjunt de serveis que es despleguen a Internet mitjançant serveis web.

Un **servei web** és una tecnologia que fa servir un conjunt de protocols i estàndards per tal d'intercanviar dades entre aplicacions.

Podeu veure els serveis web com components d'aplicacions distribuïdes que estan disponibles de forma externa i que es poden fer servir per integrar aplicacions escrites en diferents llenguatges (Java, .NET, PHP, etc.) i que s'executen en plataformes diferents (Windows, Linux, etc.). **La seva característica principal és, doncs, la interoperabilitat.**

Un servei web publica una lògica de negoci exposada com a servei als clients. La diferència més gran entre una lògica de negoci exposada com a servei web i, per exemple, una lògica de negoci exposada amb un mètode d'un EJB és que els serveis web SOAP proporcionen una interfície poc acoblada als clients. Això permet comunicar aplicacions que s'executin en diferents sistemes operatius, desenvolupades amb diferents tecnologies i amb diferents llenguatges de programació.

Spring és un bastiment (*framework*, en anglès) amb mòduls basat en Java Enterprise Edition. La principal característica del seu *core* és la utilització del patró de disseny d'inversió de control (de l'anglès *Inversion of Control*) i també la injecció de dependències (de l'anglès *Dependency Injection*), un tipus d'IoC. Spring ofereix també un conjunt de projectes/mòduls per desenvolupar serveis web SOAP i RESTful.

Els serveis web i, per tant, les aplicacions orientades a serveis es poden implementar amb diferents tecnologies. Tant Java EE 7 com Spring proporcionen implementacions per treballar amb serveis web SOAP i serveis web RESTful.

SOAP (de l'anglès *Simple Object Access Protocol*) és un protocol que permet la interacció de serveis web basats en XML. L'especificació de SOAP inclou la sintaxi amb la qual s'han de definir els missatges, les regles de codificació dels tipus de dades i les regles de codificació que regiran les comunicacions entre aquests serveis web.

L'especificació de Java EE 7 inclou diverses especificacions que donen complet suport a la creació i el consum de serveis web SOAP; la més destacada és JAX-WS (de l'anglès *Java API for XML-Based Web Services*), que utilitza missatges XML seguint el protocol SOAP i oculta la complexitat d'aquest protocol proporcionant una API senzilla per al desenvolupament, el desplegament i el consum de serveis web SOAP amb Java EE.

JAX-WS és l'API estàndard que defineix Java EE per desenvolupar i desplegar serveis web SOAP.

Spring també proporciona un projecte anomenat Spring-WS (de l'anglès *Spring Web Services*) enfocat a la creació i el consum de serveis web SOAP. Spring-WS es basa completament en Spring i porta inherent al seu model molts dels conceptes clau a Spring, com poden ser la inversió del control i la injecció de dependències.

Spring-WS és el projecte que ofereix Spring per desenvolupar i desplegar serveis web SOAP.

JAX-WS no pressuposa un model de desenvolupament per als serveis web i el desenvolupador pot triar tant una estratègia *Code First* (també anomenada *Contract Last*) com una estratègia *Contract First*, tot i que el més habitual és utilitzar l'estratègia *Code First*.

Spring-WS, en canvi, només suporta la creació de serveis web SOAP seguint una estratègia *Contract First*.

Per què 'Contract First'?

Si voleu saber les motivacions que té Spring per suportar tan sols una estratègia *Contract First* ho podeu fer consultant aquesta URL: bit.ly/2niObVu.

WSDL és un document XML que descriu un servei web SOAP. Descriu on es localitza un servei, quines operacions proporciona, el format dels missatges que han de ser intercanviats i com cal cridar el servei.

Quan parlem de serveis web, una estratègia *Code First* vol dir crear primer les classes Java que implementaran el servei i, a partir d'aquestes, generar els documents WSDL de definició del servei.

Una estratègia *Contract First* vol dir exactament el contrari: fer primer la definició del servei abans d'implementar-lo. Això vol dir descriure els paràmetres i tipus de retorn del servei amb XSD (de l'anglès *XML Schema Definitions*), després utilitzar aquest XSD per generar el document WSDL que serà el contracte públic del servei i, finalment, generar les classes Java que implementaran el servei d'acord amb aquest contracte.

L'elecció entre JAX-WS i Spring-WS a l'hora de desenvolupar serveis web SOAP amb Java no és senzilla i no hi ha cap de les dues tecnologies que sigui una clara guanyadora; alguns dels motius per utilitzar Spring-WS són:

- Per disseny, una estratègia *Contract First* promou un baix acoblament entre el contracte i la implementació del servei web.
- Suport més ampli amb les llibreries de tractament XML.
- Suport més ampli i flexible en els mapatges entre XML i objectes Java.
- Permet reutilitzar tot el coneixement que tingueu en Spring.
- Suporta WS-Security i permet integrar-ho amb Spring Security.

1.1 Escrivint un servei web SOAP de consulta de dades d'empreses amb Spring-WS

Veurem els conceptes referents a la definició de serveis web SOAP amb Spring-WS desenvolupant un servei web SOAP amb Spring-WS que permeti consultar les dades d'una empresa per CIF.

Spring-WS ens obliga a utilitzar una estratègia *Contract First* per crear serveis web SOAP, això vol dir que primer descriurem els paràmetres i tipus de retorn del servei amb XSD, després utilitzarem aquest XSD per generar el document WSDL que serà el contracte públic del servei i, finalment, generarem les classes Java que implementaran el servei d'acord amb aquest contracte.

Quan desenvolapeu un servei amb una estratègia *Contract First* us heu de centrar a **concretar el XML** que definirà el servei, el codi Java que ho implementarà passa a segon terme!

1.1.1 Creació i configuració inicial del projecte

El projecte ja té el pom.xml preparat per poder desenvolupar el servei web amb Spring-WS; concretament, s'hi ha afegit aquestes dues dependències:

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-ws</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>wsdl4j</groupId>
7   <artifactId>wsdl4j</artifactId>
8 </dependency>
```

Descarregueu el codi del projecte "Springsoapemptioc" en l'estat inicial d'aquest apartat en l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

Farem el desplegament del servei web creant una aplicació executable. Per fer el desplegament creant una aplicació executable que s'executarà al contenidor de *servlets* que Spring porta incorporat cal crear una classe Java anotada amb `@SpringBootApplication` amb un mètode `main` que cridi el mètode `run` de `SpringApplication`. Per fer-ho ja us hem creat la classe `Application` al paquet `cat.xtec.ioc.service.impl` amb el següent codi:

```

1 package cat.xtec.ioc.service.impl;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class Application {
8
9     public static void main(String[] args) {
10         SpringApplication.run(Application.class, args);
11     }
12 }
```

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

El fet d'anotar la classe amb `@SpringBootApplication` afegeix tota la configuració necessària per tal que l'aplicació pugui funcionar de forma autònoma.

1.1.2 Creació del servei web SOAP

El servei web SOAP que volem crear és molt senzill, tan sols ens ha de permetre consultar la informació d'una empresa d'un repositori d'empreses.

Com que crearem el servei web seguint una estratègia *Contract First*, el primer que ens cal fer és crear el document XSD que definirà el model de domini. El model de domini es defineix amb un fitxer d'esquema XML que després Spring-WS s'encarregarà d'exportar a WSDL.

La part principal del nostre model del domini serà l'entitat `Company`, que representarà les dades d'una empresa i la modelarem amb un document XSD. Per fer-ho creeu un fitxer anomenat `company.xsd` al directori `/src/main/resources/` amb el següent contingut:

```

1 <xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://cat.
2   xtec.ioc/domain/company"
3     targetNamespace="http://cat.xtec.ioc/domain/company"
4     elementFormDefault="qualified">
5
6   <xsccomplexType name="company">
7     <xsssequence>
8       <xselement name="cif" type="xs:string"/>
9       <xselement name="name" type="xs:string"/>
10      <xselement name="employees" type="xs:int"/>
11      <xselement name="email" type="xs:string"/>
12      <xselement name="web" type="xs:string"/>
13    </xsssequence>
14  </xsccomplexType>
15
16 </xsschema>
```

Fixeu-vos que es defineix un tipus XML complex per construir el model del domini; en aquest cas, un objecte company que conté el CIF, el nom, el nombre de treballadors, el correu i la pàgina web de l'empresa.

Un cop definida la nostra entitat del domini ens cal concretar els tipus de les peticions i les respostes del servei web que permetrà consultar la informació de l'empresa per CIF. Per fer-ho creeu un fitxer anomenat companyoperations.xsd al directori `/src/main/resources/` amb el següent contingut:

```

1 <xss:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://cat.
2   xtec.ioc/domain/company/services"
3     targetNamespace="http://cat.xtec.ioc/domain/company/services"
4     xmlns:company="http://cat.xtec.ioc/domain/company"
5     elementFormDefault="qualified">
6       <xss:import namespace="http://cat.xtec.ioc/domain/company" schemaLocation="
7         company.xsd"/>
8       <xss:element name="getCompanyRequest">
9         <xss:complexType>
10        <xss:sequence>
11          <xss:element name="cif" type="xs:string"/>
12        </xss:sequence>
13      </xss:complexType>
14    </xss:element>
15
16    <xss:element name="getCompanyResponse">
17      <xss:complexType>
18        <xss:sequence>
19          <xss:element name="company" type="company:company"/>
20        </xss:sequence>
21      </xss:complexType>
22    </xss:element>
23
24  </xss:schema>
```

Fixeu-vos que:

- S'importa la definició del tipus complex company que ha de tornar la resposta.
- Es defineix la petició com un tipus XML complex que conté la cadena on passarem el CIF de l'empresa de la qual volem consultar les dades.
- Es defineix la resposta també com un tipus XML complex que conté la informació de l'empresa consultada.

Per tal d'utilitzar els tipus que hem definit als fitxers XSD ens cal generar les classes Java per a aquests tipus. Spring-WS serà capaç de generar la classe Java Company que representa el model de domini i les classes Java que modelaran la petició (GetCompanyRequest) i la resposta (GetCompanyResponse) del servei web.

Aquesta és una part fonamental als serveis web SOAP: la conversió dels missatges SOAP d'XML cap a Java i a l'inrevés. Aquesta tasca no seria senzilla si l'haguéssiu de fer a mà, però Spring ho fa fàcil utilitzant el bastiment (*framework*, en anglès) JAXB, i ho farem en temps de construcció del projecte.

Modularitat en XSD

Es podria definir l'entitat del domini i les operacions en un únic document XSD, però és molt més modular fer-ho en documents separats i utilitzar la capacitat de fer importacions que proporciona XSD.

JAXB (de l'anglès *Java Architecture for XML Binding*) és un bastiment que permet fer la conversió entre documents XML i objectes Java, i a l'inrevés.

Això es pot fer de diverses formes; una manera senzilla és que ho faci el *plugin* de Maven jaxb-maven-plugin en temps de construcció del projecte. Afegiu les següents línies al fitxer pom.xml:

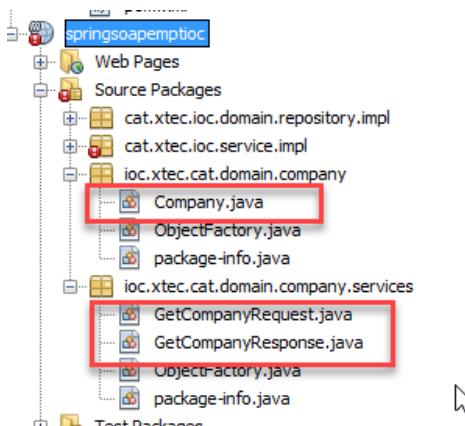
```

1 <plugin>
2   <groupId>org.codehaus.mojo</groupId>
3   <artifactId>jaxb2-maven-plugin</artifactId>
4   <version>1.6</version>
5   <executions>
6     <execution>
7       <id>xjc</id>
8       <goals>
9         <goal>xjc</goal>
10      </goals>
11    </execution>
12  </executions>
13  <configuration>
14    <schemaDirectory>${project.basedir}/src/main/resources/<
15      schemaDirectory>
16    <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
17    <clearOutputDir>false</clearOutputDir>
18  </configuration>
19 </plugin>

```

Si recarregueu el pom.xml fent clic amb el botó dret damunt el nom del projecte i prement l'opció *Reload POM* del menú contextual i després feu *Clean and Build*, també al menú contextual de NetBeans, veureu com a la fase de generació el *plugin* xjc ha generat, a partir dels documents XSD de definició de servei, la classe Java que modela el domini i les classes Java que modelen la petició i la resposta del servei web (vegeu la figura 3.1).

FIGURA 1.1. Classes Java generades



```

1 — jaxb2-maven-plugin:1.6:xjc (xjc) @ springsoapempioc —
2 Generating source...
3 Analizando un esquema...
4 Compilando un esquema...
5 ioc\xtec\cat\domain\company\services\GetCompanyRequest.java
6 ioc\xtec\cat\domain\company\services\GetCompanyResponse.java
7 ioc\xtec\cat\domain\company\services\ObjectFactory.java
8 ioc\xtec\cat\domain\company\services\package-info.java
9 ioc\xtec\cat\domain\company\Company.java
10 ioc\xtec\cat\domain\company\ObjectFactory.java
11 ioc\xtec\cat\domain\company\package-info.java

```

Ara toca crear el repositori d'on el servei ha de consultar la informació de les empreses. En el nostre cas, per simplicitat, farem servir un repositori *in memory*

que tindrà una llista precarregada d'empreses i un mètode `findCompany` que permet consultar una de les empreses per CIF. Òbviament, en un projecte real la definició del servei serà molt més complexa!

Per fer-ho creeu una classe Java anomenada `InMemoryCompanyRepository` al paquet `cat.xtec.ioc.domain.repository.impl` amb el següent codi:

```

1 package cat.xtec.ioc.domain.repository.impl;
2
3 import ioc.xtec.cat.domain.company.Company;
4 import java.util.ArrayList;
5 import java.util.List;
6 import javax.annotation.PostConstruct;
7 import org.springframework.stereotype.Repository;
8 import org.springframework.util.Assert;
9
10 @Repository
11 public class InMemoryCompanyRepository {
12
13     private static final List<Company> companies = new ArrayList<Company>();
14
15     @PostConstruct
16     public void initData() {
17         Company oracle = new Company();
18         oracle.setCif("XXXXXXXX");
19         oracle.setName("Oracle");
20         oracle.setEmployees(5000);
21         oracle.setEmail("oracle@oracle.com");
22         oracle.setWeb("http://www.oracle.com");
23
24         companies.add(oracle);
25
26         Company ms = new Company();
27         ms.setCif("YYYYYYYY");
28         ms.setName("Microsoft");
29         ms.setEmployees(10000);
30         ms.setEmail("microsoft@microsoft.com");
31         ms.setWeb("http://www.microsoft.com");
32
33         companies.add(ms);
34
35         Company redhat = new Company();
36         redhat.setCif("ZZZZZZZZ");
37         redhat.setName("Red Hat");
38         redhat.setEmployees(2000);
39         redhat.setEmail("redhat@redhat.com");
40         redhat.setWeb("http://www.redhat.com");
41         companies.add(redhat);
42     }
43
44     public Company findCompany(String cif) {
45         Assert.notNull(cif);
46         Company result = null;
47         for (Company company : companies) {
48             if (cif.equals(company.getCif())) {
49                 result = company;
50             }
51         }
52
53         return result;
54     }
55 }
```

Amb tot això ja estem preparats per crear l'*endpoint* del servei web SOAP que tornarà la informació de les empreses. Noteu que, per sota, hi ha un *servlet* que s'encarrega de recollir les peticions SOAP que vagin al servei i les envia a l'*endpoint* per tal de processar-les.

Creeu una classe Java anomenada CompanyEndpoint al paquet cat.xtec.ioc.service.impl amb el següent codi:

```

1  package cat.xtec.ioc.service.impl;
2
3  import cat.xtec.ioc.domain.repository.impl.InMemoryCompanyRepository;
4  import ioc.xtec.cat.domain.company.services.GetCompanyRequest;
5  import ioc.xtec.cat.domain.company.services.GetCompanyResponse;
6
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.ws.server.endpoint.annotation.Endpoint;
9  import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
10 import org.springframework.ws.server.endpoint.annotation.RequestPayload;
11 import org.springframework.ws.server.endpoint.annotation.ResponsePayload;
12
13 @Endpoint
14 public class CompanyEndpoint {
15
16     private static final String NAMESPACE_URI = "http://cat.xtec.ioc/domain/
17         company/services";
18
19     private InMemoryCompanyRepository companyRepository;
20
21     @Autowired
22     public CompanyEndpoint(InMemoryCompanyRepository companyRepository) {
23         this.companyRepository = companyRepository;
24     }
25
26     @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getCompanyRequest")
27     @ResponsePayload
28     public GetCompanyResponse getCompany(@RequestPayload GetCompanyRequest
29                                         request) {
30         GetCompanyResponse response = new GetCompanyResponse();
31         response.setCompany(companyRepository.findCompany(request.getCif()));
32
33         return response;
34     }
35 }
```

Fixeu-vos que:

- Hem injectat el repositori d'empreses al constructor del servei amb Spring.
- Hem anotat la classe amb `@Endpoint` per indicar a Spring que aquesta classe correspon a un *endpoint* d'un servei web SOAP i tindrà mètodes que serviran peticions SOAP. `@Endpoint` és una versió especialitzada de l'anotació `@Component`.
- TARGET_NAMESPACE representa l'espai de noms que heu definit anteriorment al document XSD. Es fa servir per mapar les peticions a mètodes específics de l'*endpoint*.
- Hem anotat el mètode `getCompany` amb l'anotació `@PayloadRoot` on es defineix l'espai de noms i el mètode que servirà les peticions de cerca d'empreses per CIF.
- El missatge SOAP que consulti una empresa per CIF haurà de fer referència al `PayloadRoot` especificat.
- Hem anotat el mètode amb `@ResponsePayload` per indicar que el mètode tornarà un objecte de tipus `GetCompanyResponse` amb la informació de l'empresa. Spring s'encarregarà de convertir aquest objecte a XML.

- El paràmetre del mètode l'hem anotat amb `@RequestPayload` per indicar que el missatge d'entrada serà de tipus `GetCompanyRequest`. Spring s'encarregarà de convertir el missatge XML d'entrada a un objecte d'aquest tipus.
- Tant el tipus de retorn com el paràmetre corresponen a classes generades automàticament en el procés de construcció a partir de la definició del fitxer d'esquema XML `companyoperations.xsd`.

Un cop creat l'*endpoint* del servei web SOAP tan sols ens queda configurar Spring per carregar tota la configuració. Això ho podeu fer de diverses maneres, ja sigui amb fitxers de configuració o amb anotacions a classes Java de configuració. A l'exemple utilitzarem aquesta darrera tècnica.

Creeu una classe Java anomenada `WebServiceConfig` al paquet `cat.xtec.ioc.service.impl` amb el següent codi:

```

1 package cat.xtec.ioc.service.impl;
2
3 import org.springframework.boot.web.servlet.ServletRegistrationBean;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.ComponentScan;
7 import org.springframework.context.annotation.Configuration;
8 import org.springframework.core.io.ClassPathResource;
9 import org.springframework.ws.config.annotation.EnableWs;
10 import org.springframework.ws.config.annotation.WsConfigurerAdapter;
11 import org.springframework.ws.transport.http.MessageDispatcherServlet;
12 import org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition;
13 import org.springframework.xml.xsd.SimpleXsdSchema;
14 import org.springframework.xml.xsd.XsdSchema;
15
16 @EnableWs
17 @Configuration
18 @ComponentScan("cat.xtec.ioc.service.impl, cat.xtec.ioc.domain.repository.impl")
19 public class WebServiceConfig extends WsConfigurerAdapter {
20
21     @Bean
22     public ServletRegistrationBean messageDispatcherServlet(ApplicationContext
23         applicationContext) {
24         MessageDispatcherServlet servlet = new MessageDispatcherServlet();
25         servlet.setApplicationContext(applicationContext);
26         servlet.setTransformWSDLLocations(true);
27         return new ServletRegistrationBean(servlet, "/soapws/*");
28     }
29
30     @Bean(name = "companyoperations")
31     public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema
32         companiesSchema) {
33         DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition
34             ();
35         wsdl11Definition.setPortTypeName("CompaniesPort");
36         wsdl11Definition.setLocationUri("/soapws");
37         wsdl11Definition.setTargetNamespace("http://cat.xtec.ioc/domain/company
38             /services");
39         wsdl11Definition.setSchema(companiesSchema);
40         return wsdl11Definition;
41     }
42
43     @Bean
44     public XsdSchema companiesSchema() {
45         return new SimpleXsdSchema(new ClassPathResource("companyoperations.xsd
46             "));
47     }

```

```

43
44     @Bean(name = "company")
45     public XsdSchema companySchema() {
46         return new SimpleXsdSchema(new ClassPathResource("company.xsd"));
47     }
48 }
```

Fixeu-vos que:

- Spring fa servir un *servlet* especial de tipus `MessageDispatcherServlet` per servir peticions SOAP, i cal que el registreu al `ApplicationContext` de l'aplicació.
- El mètode `defaultWsdl11Definition` s'encarrega de configurar el document WSDL de definició del servei web a partir del document XSD especificat.
- L'ús de `DefaultWsdl11Definition` permet la generació automàtica del document WSDL.
- El mètode `defaultWsdl11Definition` també defineix l'URI i l'espai de noms que caldrà que utilitzeu per cridar el servei web i que estarà disponible en el document WSDL de definició del servei.

Aquesta és tota la feina que ens cal fer per implementar el servei web de consulta de dades d'empreses; ara tan sols ens manca fer el desplegament i provar-lo.

1.1.3 Desplegament del servei web SOAP

Un cop creat el servei web cal que el desplegueu per tal de fer-lo accessible als clients.

El desplegament del servei web es pot fer de diverses maneres, entre les quals:

- Desplegant l'aplicació Java EE que el conté mitjançant els mecanismes normals de desplegament de qualsevol aplicació Java EE.
- Creant una aplicació executable que s'executarà amb un contenidor de *servlets* Tomcat que Spring porta incorporat.

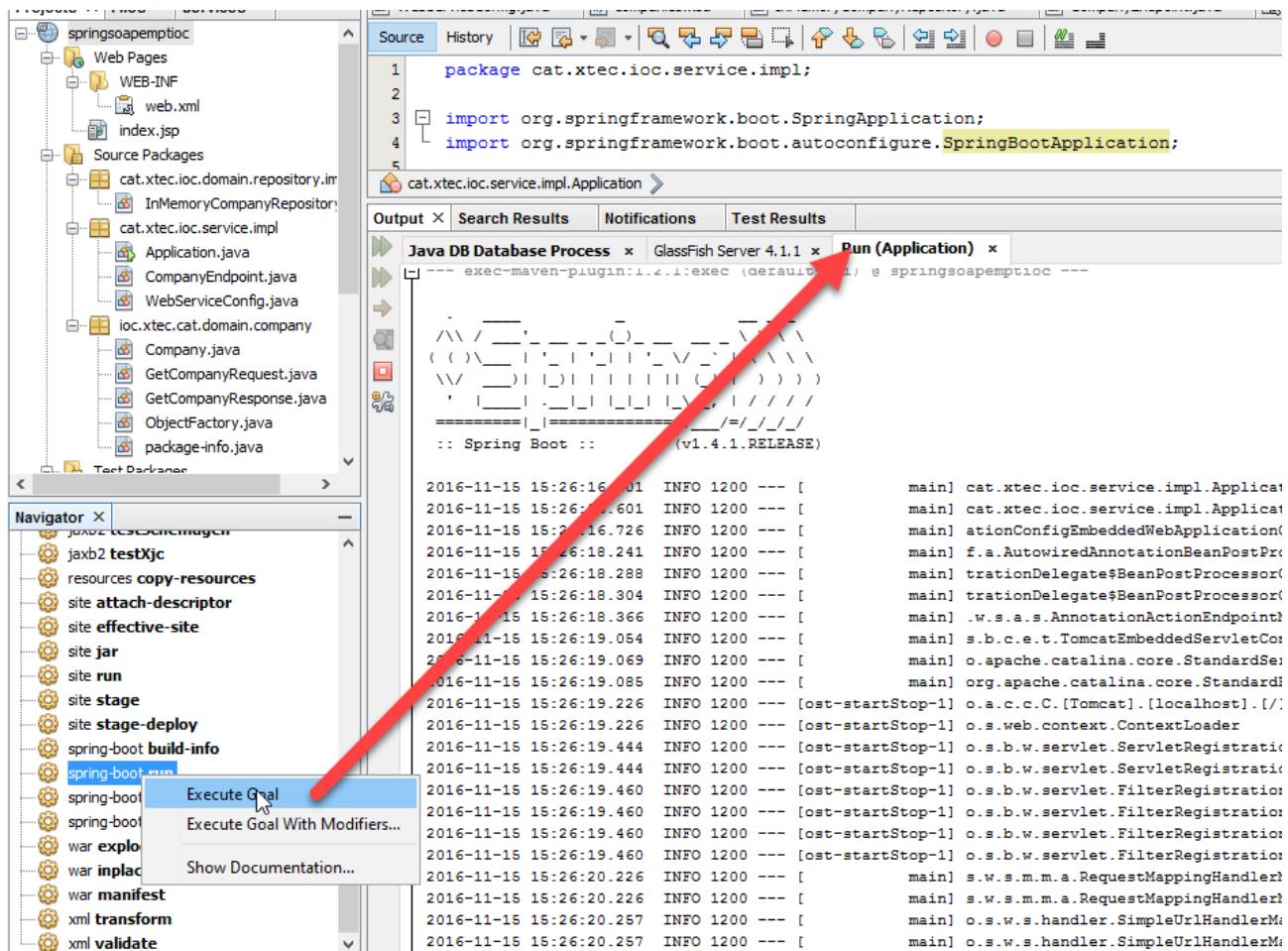
Prerequisits per a l'execució

Abans d'executar la classe cal que us assegureu de tenir el servidor Glassfish parat, ja que, si no, no podrà arrencar el servei Tomcat on desplegarem l'aplicació per un conflicte amb els ports. Tant Glassfish com Tomcat estan configurats per utilitzar el port 8080 per defecte.

Farem el desplegament del servei web creant una aplicació executable. Per fer el desplegament creant una aplicació executable que s'executarà al contenidor de *servlets* que Spring porta incorporat tan sols ens cal una classe Java anotada amb `@SpringBootApplication` amb un mètode `main` que cridi el mètode `run` de `SpringApplication`. Al projecte inicial ja teniu una classe `Application` al paquet `cat.xtec.ioc.service.impl` que permet que l'aplicació pugui funcionar de forma autònoma.

Poseu-vos damunt la classe Application, feu clic amb el botó dret del ratolí i seleccioneu l'opció *Run* (també podeu executar el *goal* de Maven `spring-boot:run`). Veureu que apareix una finestra a NetBeans amb l'execució de l'aplicació amb Spring (vegeu la figura 3.2).

FIGURA 1.2. Execució de l'aplicació amb el servei web



Podeu provar que el desplegament ha anat bé consultant el document WSDL generat a l'enllaç localhost:8080/soapws/companyoperations.wsdl amb qualsevol navegador, i heu de veure el document WSDL de definició del servei generat:

```

1  <wsdl:definitions targetNamespace="http://cat.xtec.ioc/domain/company/services"
2    >
3    <wsdl:types></wsdl:types>
4    <wsdl:message name="getCompanyRequest"></wsdl:message>
5    <wsdl:message name="getCompanyResponse"></wsdl:message>
6    <wsdl:portType name="CompaniesPort"></wsdl:portType>
7    <wsdl:binding name="CompaniesPortSoap11" type="tns:CompaniesPort"></
8      wsdl:binding>
9    <wsdl:service name="CompaniesPortService"></wsdl:service>
10   </wsdl:definitions>
```

1.1.4 Prova del servei web

Si tot ha anat bé ja teniu el servei web desplegat i a punt per provar-lo. Però com ho fem? Quin format han de tenir els missatges?

Quan es desplega el servei web SOAP es genera un fitxer WSDL de definició del servei web. El fitxer WSDL està forma per elements XML que descriuen completament el servei web i com s'ha de consumir.

El fitxer WSDL té una secció abstracta per definir els ports, els missatges i els tipus de dades de les operacions, i una part concreta que defineix la instància on hi ha aquestes operacions. Aquesta estructura permet reutilitzar la part abstracta del document.

A la part abstracta hi tenim els següents elements:

- *types*
- *message*
- *portType*

Els tipus de dades, tant d'entrada com de sortida, de les operacions es defineixen amb XSD a l'etiqueta `<types>`:

```

1  <wsdl:types>
2      <xss:schema elementFormDefault="qualified" targetNamespace="http://cat.xtec.
3          ioc/domain/company/services">
4          <xss:import namespace="http://cat.xtec.ioc/domain/company"
5              schemaLocation="company.xsd"/>
6          <xss:element name="getCompanyRequest">
7              <xss:complexType>
8                  <xss:sequence>
9                      <xss:element name="cif" type="xs:string"/>
10                 </xss:sequence>
11             </xss:complexType>
12         </xss:element>
13         <xss:element name="getCompanyResponse">
14             <xss:complexType>
15                 <xss:sequence>
16                     <xss:element name="company" type="company:company"/>
17                 </xss:sequence>
18             </xss:complexType>
19         </xss:element>
20     </xss:schema>
21 </wsdl:types>

```

Per exemple, si us hi fixeu, defineix que les peticions a l'operació `getCompanyRequest` reben un paràmetre de tipus `string` i a les respostes es torna un tipus de dades complex anomenat `company` que està format per un *int* i quatre `string` (`cif`, `name`, `employees`, `email` i `web`).

La definició del tipus complex `company` s'importa d'un fitxer XSD separat que té la definició del model del domini.

```

1 <xss:schema elementFormDefault="qualified" targetNamespace="http://cat.xtec.ioc/
2   domain/company">
3   <xss:complexType name="company">
4     <xss:sequence>
5       <xss:element name="cif" type="xs:string"/>
6       <xss:element name="name" type="xs:string"/>
7       <xss:element name="employees" type="xs:int"/>
8       <xss:element name="email" type="xs:string"/>
9       <xss:element name="web" type="xs:string"/>
10    </xss:sequence>
11  </xss:complexType>
12 </xss:schema>
```

Després de la definició dels tipus de dades trobem els elements <message> amb la definició dels missatges que es poden intercanviar les operacions del servei web, amb una entrada per a la petició i una altra per a la resposta:

```

1 <wsdl:message name="getCompanyRequest">
2   <wsdl:part element="tns:getCompanyRequest" name="getCompanyRequest">
3   </wsdl:part>
4 </wsdl:message>
5 <wsdl:message name="getCompanyResponse">
6   <wsdl:part element="tns:getCompanyResponse" name="getCompanyResponse">
7   </wsdl:part>
8 </wsdl:message>
```

I després, els elements <portType> ens defineixen les operacions que es poden fer al servei web amb els seus paràmetres i el tipus de retorn:

```

1 <wsdl:portType name="CompaniesPort">
2   <wsdl:operation name="getCompany">
3     <wsdl:input message="tns:getCompanyRequest" name="getCompanyRequest">
4     </wsdl:input>
5     <wsdl:output message="tns:getCompanyResponse" name="getCompanyResponse">
6     </wsdl:output>
7   </wsdl:operation>
8 </wsdl:portType>
```

Veiem, per exemple, que l'operació getCompanyRequest rep com a paràmetre d'entrada un tns:getCompanyRequest i que retorna un tns:getCompanyResponse. Aquests tipus de dades han estat definides completament en el document XSD que hem creat per definir el servei.

A la part concreta hi tenim els següents elements:

- binding
- service

L'element <binding> defineix el protocol i el format en què es poden fer les operacions; en el nostre cas, les operacions es faran per HTTP i amb un estil de *Document*.

```

1 <wsdl:binding name="CompaniesPortSoap11" type="tns:CompaniesPort">
2   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/
3     http"/>
4   <wsdl:operation name="getCompany">
```

```

4      <soap:operation soapAction="" />
5      <wsdl:input name="getCompanyRequest">
6          <soap:body use="literal" />
7      </wsdl:input>
8      <wsdl:output name="getCompanyResponse">
9          <soap:body use="literal" />
10     </wsdl:output>
11 </wsdl:operation>
12 </wsdl:binding>
```

L'element `<service>`, per la seva part, defineix el lloc físic on hi ha el servei web (adreça URL).

```

1 <wsdl:service name="CompaniesPortService">
2     <wsdl:port binding="tns:CompaniesPortSoap11" name="CompaniesPortSoap11">
3         <soap:address location="http://localhost:8080/soapws" />
4     </wsdl:port>
5 </wsdl:service>
```

En el nostre cas, el servei web es a l'URL localhost:8080/soapws.

Amb tot això ja sabem el format dels missatges SOAP que cal enviar per accedir al servei web i on cal enviar-los; creeu un fitxer anomenat `request.xml`, que representarà el missatge SOAP de petició, amb el següent contingut:

```

1 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" 
2     xmlns:gs="http://cat.xtec.ioc/domain/company/services">
3     <soapenv:Header/>
4     <soapenv:Body>
5         <gs:getCompanyRequest>
6             <gs:cif>XXXXXXXXXX</gs:cif>
7         </gs:getCompanyRequest>
8     </soapenv:Body>
9 </soapenv:Envelope>
```

I ja tan sols ens queda fer una petició al servei web per provar-ho. La nostra proposta és que feu servir cURL, que és una eina molt útil per fer peticions HTTP de diferents tipus i que és multiplataforma. Si feu servir Linux possiblement ja la tingueu instal·lada al sistema, i en cas que utilitzeu Windows la podeu descarregar del següent enllaç: curl.haxx.se/download.html.

La sintaxi de cURL per fer peticions és molt senzilla; per exemple, per consultar les dades de l'empresa que té per CIF XXXXXXXXX feu un command prompt:

```

1 curl --header "content-type: text/xml" -d @request.xml http://localhost:8080/
    soapws/
```

Indiquem que la petició s'envia al fitxer `request.xml` amb el tipus MIME `text/xml`. Aquest fitxer és el que acabeu de crear amb el missatge SOAP de petició, i l'URL és l'URL on escolta el servei web.

Si executeu la comanda anterior i tot ha anat correctament, veureu la resposta del servei web amb les dades de l'empresa que té per CIF XXXXXXXXX:

```

1 <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
2     <SOAP-ENV:Header/>
3     <SOAP-ENV:Body>
4         <ns3:getCompanyResponse xmlns:ns2="http://cat.xtec.ioc/domain/company"
            xmlns:ns3="http://cat.xtec.ioc/domain/company/services">
```

```

5   <ns3:company>
6     <ns2:cif>XXXXXXXX</ns2:cif>
7     <ns2:name>Oracle</ns2:name>
8     <ns2:employees>5000</ns2:employees>
9     <ns2:email>oracle@oracle.com</ns2:email>
10    <ns2:web>http://www.oracle.com</ns2:web>
11  </ns3:company>
12 </ns3:getCompanyResponse>
13 </SOAP-ENV:Body>
14 </SOAP-ENV:Envelope>
```

I això és tot! Hem vist els passos que us cal fer per desenvolupar, desplegar i provar un servei web SOAP amb Spring-WS seguint una estratègia *Contract First*.

1.2 Fent servir la consulta de dades d'empreses des d'una aplicació Java 'stand-alone'

Crearem una aplicació Java que consumeixi el servei web de consulta de dades d'empreses i ho farem desenvolupant un client Java *stand-alone* que accedirà al servei web amb Spring-WS.

Els passos generals per fer un client amb Spring-WS són els següents:

1. Generar els artefactes necessaris per poder consumir el servei web a partir de la definició del servei en format WSDL.
2. Codificar la classe que farà la crida al servei web heretant de la classe `WebServiceGatewaySupport` que proporciona Spring-WS.
3. Compilar i executar el client.

El procediment que expliquem és **genèric i serveix per a qualsevol servei web SOAP**, estiguí o no codificat amb Spring-WS. Només ens cal la definició WSDL del servei.

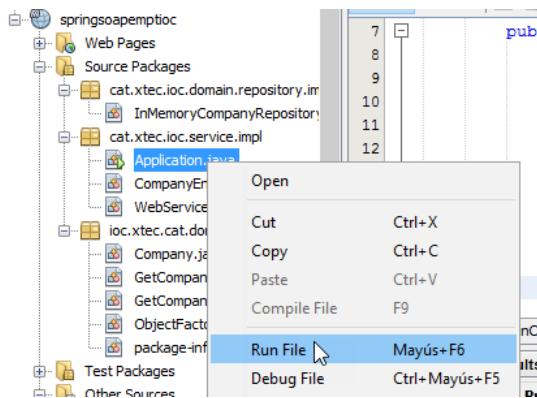
El client que farem consultarà les dades de les empreses d'un servei web SOAP desenvolupat amb Spring-WS. Descarregueu el codi del servei web que consultarem en el següent i importeu-lo a NetBeans.

Quan tingueu el projecte importat a NetBeans cal que desplegueu el servei web que conté; per exemple, creant una aplicació executable que s'executarà amb un contingidor de *servlets* Tomcat que Spring porta incorporat. Ho podeu fer de diverses maneres, una és fent clic amb el botó dret damunt de la classe Application del paquet `cat.xtec.ioc.client` i seleccionant *Run File* (vegeu la figura 1.3).

Prerequisits per a l'execució

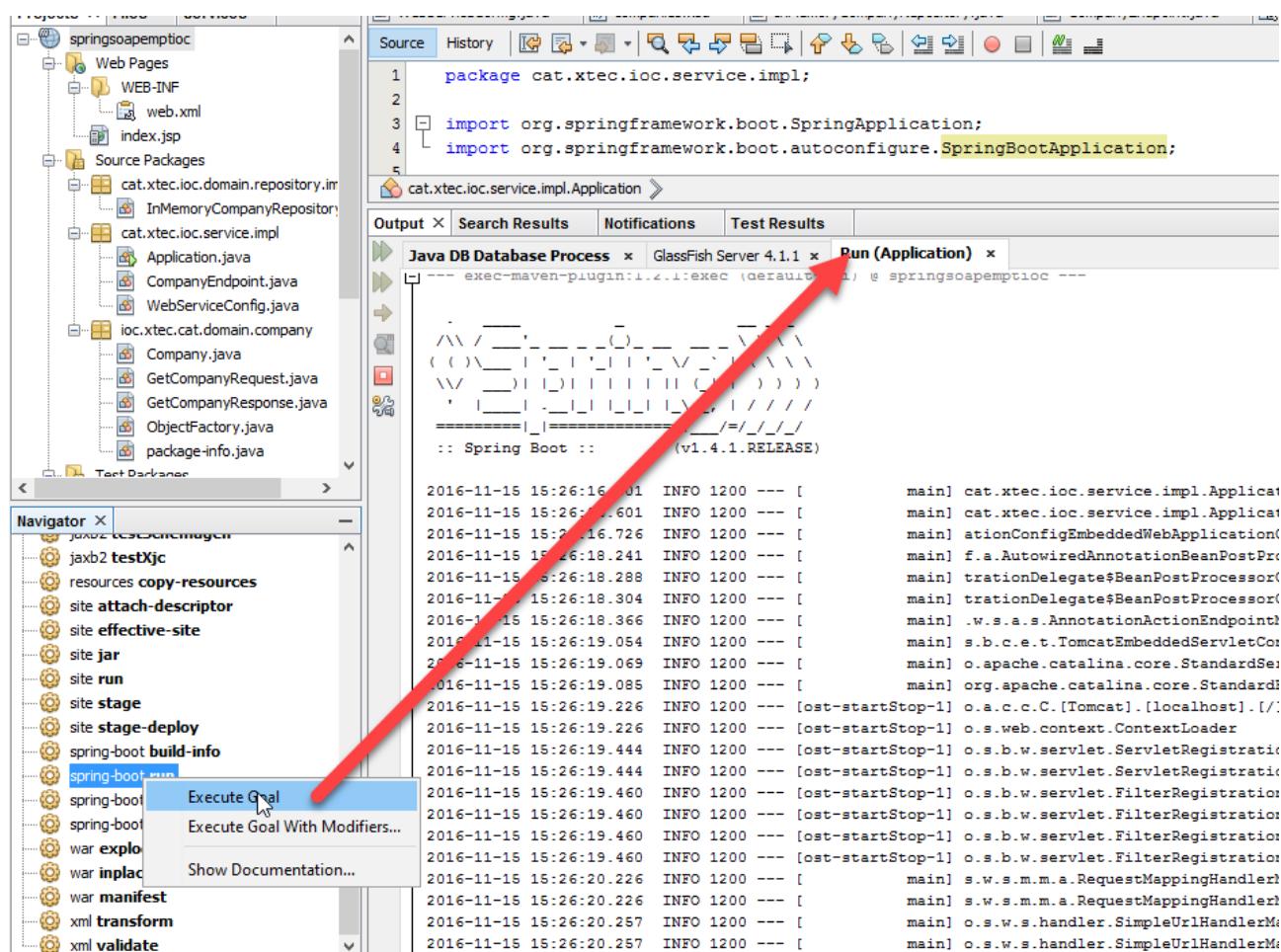
Abans d'executar la classe cal que us assegureu de tenir el servidor Glassfish parat, ja que, si no, no podrà arrencar el servei Tomcat on desplegarem l'aplicació per un conflicte amb els ports, ja que tant Glassfish com Tomcat estan configurats per utilitzar el port 8080 per defecte.

FIGURA 1.3. Execució del servei web SOAP de consulta d'empreses



També ho podeu fer executant el *goal* de Maven `spring-boot:run`. Amb qualsevol de les dues opcions veureu que apareix una finestra a NetBeans amb l'execució de l'aplicació amb Spring (vegeu la figura 1.4).

FIGURA 1.4. Finestra d'execució del servei web SOAP de consulta d'empreses



Podeu provar que el desplegament ha anat bé consultant el document WSDL generat a l'enllaç localhost:8080/soapws/companyoperations.wsdl amb qualsevol navegador, i heu de veure el document WSDL de definició del servei.

Després de desplegar el servei web que volem consultar ens cal crear el client Java que el consultarà amb Spring-WS.

Descarregueu el codi del projecte "Springsoapemtioic" en l'estat inicial d'aquest apartat en l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

El primer que cal fer és generar els artefactes necessaris per poder consumir el servei web des d'aquest client a partir de la seva definició en format WSDL. Això es pot fer de diverses formes, i una manera senzilla és que ho faci un *plugin* de Maven en temps de construcció del projecte.

Afegiu les següents línies al fitxer pom.xml i feu un *Reload POM* del projecte:

```

1 <plugin>
2   <groupId>org.jvnet.jaxb2.maven2</groupId>
3   <artifactId>maven-jaxb2-plugin</artifactId>
4   <version>0.13.1</version>
5   <executions>
6     <execution>
7       <goals>
8         <goal>generate</goal>
9       </goals>
10      </execution>
11    </executions>
12    <configuration>
13      <schemaLanguage>WSDL</schemaLanguage>
14      <generatePackage>cat.xtec.ioc.domain.companies.wsdl</generatePackage>
15      <schemas>
16        <schema>
17          <url>http://localhost:8080/soapws/companyoperations.wsdl</url>
18        </schema>
19      </schemas>
20    </configuration>
21 </plugin>
```

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

Aquest *plugin* generarà automàticament les classes necessàries per consultar el servei web de consulta de dades d'empreses en temps de compilació i les deixarà al paquet `cat.xtec.ioc.domain.companies.wsdl`. Fixeu-vos també que estem especificant en la configuració del *plugin* la localització del document WSDL amb la definició del servei.

Si feu clic amb el botó dret damunt el projecte “Springsoapempclientioc” i feu *Clean and Build* veureu que el *plugin* genera, entre d'altres, les següents classes a partir del document WSDL de definició de servei:

- `Company.java`
- `GetCompanyRequest.java`
- `GetCompanyResponse.java`

Per tal que el projecte funcioni cal que l'execueu amb JDK 1.8; podeu canviar el JDK a les propietats del projecte, a l'apartat *Build / Compile / Java Platform*.

Un cop generats els artefactes necessaris cal que creeu la classe Java que farà la crida al servei web. Creeu-la, per exemple, al paquet `cat.xtec.ioc.client` i anomeneu-la `CompaniesClient`:

```

1 package cat.xtec.ioc.client;
2
3 import cat.xtec.ioc.domain.companies.wsdl.GetCompanyRequest;
4 import cat.xtec.ioc.domain.companies.wsdl.GetCompanyResponse;
5 import org.springframework.ws.client.core.support.WebServiceGatewaySupport;
6 import org.springframework.ws.soap.client.core.SoapActionCallback;
7
8 public class CompaniesClient extends WebServiceGatewaySupport {
9
10    public GetCompanyResponse getCompanyInformation(String cif) {
11        GetCompanyRequest request = new GetCompanyRequest();
```

```

12         request.setCif(cif);
13         GetCompanyResponse response = (GetCompanyResponse)
14             getWebServiceTemplate()
15                 .marshalSendAndReceive(request,
16                     new SoapActionCallback("http://localhost:8080/soapws/
17                         getCompanyResponse"));
18
19     }
20 }
```

Aquesta classe hereta d'una classe que proporciona Spring anomenada `WebServiceGatewaySupport` i utilitza les classes generades per fer una crida al servei web mitjançant un *template* que proporciona la seva classe base.

També ens cal crear una classe de configuració per indicar a Spring que faci servir JAXB per fer les transformacions de missatge SOAP a objecte Java, i a la inversa. Creeu aquesta classe Java també al paquet `cat.xtec.ioc.client` i anomeneu-la `Client AppConfig`:

```

1 package cat.xtec.ioc.client;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.oxm.jaxb.Jaxb2Marshaller;
6
7 @Configuration
8 public class ClientAppConfig {
9
10     @Bean
11     public Jaxb2Marshaller marshaller() {
12         Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
13         marshaller.setContextPath("cat.xtec.ioc.domain.companies.wsdl");
14         return marshaller;
15     }
16
17     @Bean
18     public CompaniesClient companiesClient(Jaxb2Marshaller marshaller) {
19         CompaniesClient client = new CompaniesClient();
20         client.setDefaultUri("http://localhost:8080/soapws/companies.wsdl");
21         client.setMarshaller(marshaller);
22         client.setUnmarshaller(marshaller);
23         return client;
24     }
25 }
```

Finalment, creeu una classe Java, també al paquet `cat.xtec.ioc.client`, i l'anomeneu `RunCompaniesClient`, amb el següent codi:

```

1 package cat.xtec.ioc.client;
2
3 import cat.xtec.ioc.domain.companies.wsdl.GetCompanyResponse;
4 import org.springframework.context.annotation.
4     AnnotationConfigApplicationContext;
5
6 public class RunCompaniesClient {
7
8     public static void main(String[] args) {
9         AnnotationConfigApplicationContext ctx = new
10             AnnotationConfigApplicationContext();
11         ctx.register(ClientAppConfig.class);
12         ctx.refresh();
13         CompaniesClient companiesClient = ctx.getBean(CompaniesClient.class);
```

```

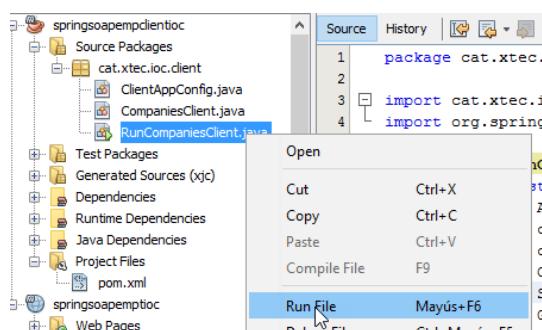
13     System.out.println("For Company XXXXXXXX");
14     GetCompanyResponse response = companiesClient.getCompanyInformation("XXXXXXX");
15     System.out.println("CIF: " + response.getCompany().getCif());
16     System.out.println("Name: " + response.getCompany().getName());
17     System.out.println("Num. employees: " + response.getCompany().
18         getEmployees());
19     System.out.println("Email: " + response.getCompany().getEmail());
20     System.out.println("Web: " + response.getCompany().getWeb());
21 }
}

```

Aquesta classe té un mètode `main` que s'encarrega de crear un objecte de tipus `CompaniesClient` amb la configuració especificada a `ClientAppConfig`, fer una petició al servei web de consulta de dades d'empreses per recuperar les dades de l'empresa que té el CIF XXXXXXXX amb aquesta classe i mostrar els resultats per a la sortida estàndard.

Ara ja podem executar el client; per fer-ho, poseu-vos damunt de la classe `RunCompaniesClient` i feu *Run File* a NetBeans (vegeu la figura 2.4).

FIGURA 1.5. Execució del client del servei web SOAP de consulta d'empreses



I obtindreu per consola les dades de l'empresa que té el CIF XXXXXXXX:

```

1 CIF: XXXXXXXX
2 Name: Oracle
3 Num. employees: 5000
4 Email: oracle@oracle.com
5 Web: http://www.oracle.com

```

1.3 Què s'ha après?

Heu vist les bases per al desenvolupament dels serveis web SOAP amb Spring-WS i les heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après:

- Les nocions bàsiques dels serveis web SOAP amb Spring-WS.
- La diferència entre *Contract First* i *Code First*.

- Les diferències bàsiques entre JAX-WS i Spring-WS.
- Desenvolupar, desplegar i provar un servei web SOAP amb Spring-WS.
- Desenvolupar i provar un client Java que consulti un servei web SOAP mitjançant Spring-WS.

2. Serveis web RESTful amb Spring. Escrivint serveis web

Explicarem, mitjançant exemples, els conceptes mes rellevants dels serveis web RESTful (de l'anglès *REpresentational State Transfer*) amb Spring MVC; aprenreu a crear-ne, a fer-ne el desplegament i a provar-los mitjançant exemples.

REST no és un protocol, sinó un conjunt de regles i principis que permeten desenvolupar serveis web fent servir HTTP com a protocol de comunicacions entre el client i el servei web, i es basa a definir **accions sobre recursos** mitjançant l'ús dels mètodes GET, POST, PUT i DELETE, inherents d'HTTP.

Per a REST, qualsevol cosa que es pugui identificar amb un URI (de l'anglès *Uniform Resource Identifier*) es considera un recurs i, per tant, es pot manipular mitjançant accions (també anomenades *verbs*) especificades a la capçalera HTTP de les peticions seguint el següent conjunt de regles i principis que regeixen REST:

- POST: crea un recurs nou.
- GET: consulta el recurs, n'obté la representació.
- DELETE: esborra un recurs.
- PUT: modifica un recurs.
- HEAD: obté metainformació del recurs.

REST es basa en l'ús d'estàndards oberts en totes les seves parts; així, fa servir URI per a la localització de recursos, HTTP com a protocol de transport, els verbs HTTP per especificar les accions sobre els recursos i els tipus MIME per a la representació dels recursos (XML, JSON, XHTML, HTML, PDF, GIF, JPG, PNG, etc.).

L'especificació de Java EE 7 inclou l'API de JAX-RS (de l'anglès *Java API for RESTful Web Services*), que fa servir un conjunt d'anotacions per simplificar el desenvolupament, el desplegament i el consum de serveis web RESTful.

JAX-RS (de l'anglès *Java API for RESTful Web Services*) és l'API que inclou l'especificació de Java EE 7 per crear i consumir serveis web basats en REST.

Spring, per la seva banda, no proporciona un projecte ni un mòdul específic per a la creació i el consum de serveis web RESTful, sinó que ho engloba com una capacitat més dins el projecte Spring MVC.

Format JSON

JSON (de l'anglès *Java Script Object Notation*) és un format lleuger d'intercanvi de dades. És facil de llegir i escriure per als éssers humans i, per a les màquines, d'analitzar i generar. Això el fa ideal per representar els recursos en arquitectures REST. Un dels principals problemes dels serveis web basats en SOAP és la mida dels missatges d'intercanvi; l'ús de JSON permet minimitzar la informació a enviar.

Spring MVC és el mòdul del projecte Spring que proporciona suport per crear i consumir serveis web basats en REST.

JAX-RS és una API centrada únicament i exclusivament en el desenvolupament de serveis web RESTful. Spring MVC és un mòdul del projecte Spring centrat en el desenvolupament web en general, i és en aquest marc que dóna complet suport tant al desenvolupament d'aplicacions web com al desenvolupament de serveis web RESTful.

Les capacitats que ofereix Spring MVC per desenvolupar serveis web RESTful són una continuació del model més general de programació que té Spring MVC. No hi ha, doncs, un *framework* o mòdul específic a Spring per desenvolupar serveis web RESTful.

Per escriure un servei web RESTful, tant amb JAX-RS com amb Spring, tan sols us caldrà un client i un servidor que es puguin comunicar per HTTP, però hauríeu de fer a mà tota la configuració, el parseig de les peticions segons el verb HTTP emprat, el mapatge entre el format de dades de la petició i els objectes Java que representen el domini i enviar les respostes amb el tipus MIME que s'especifiqui a la capçalera de la petició. Tant JAX-RS com Spring us estalvién tota aquesta feina proporcionant-vos un petit conjunt d'anotacions que us permetran desenvolupar còmodament serveis web RESTful.

Ens centrarem en les capacitats que proporciona Spring, i més concretament el mòdul Spring MCV, per donar suport a la creació i el consum de serveis web RESTful.

2.1 Un servei web RESTful que contesta ""Hello, World!!!"

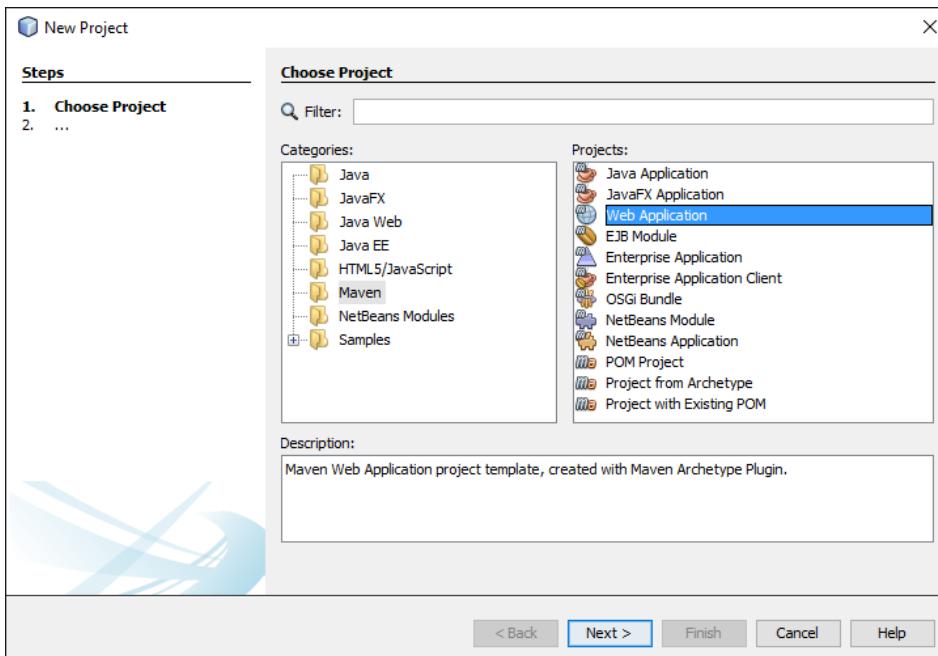
Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

Veurem els diferents conceptes bàsics dels serveis web RESTful amb el típic exemple que sempre trobeu als manuals. Farem un "Hello, World!!!" i el publicarem com a servei web RESTful utilitzant Spring MVC per fer-ho.

El servei web que farem tornarà "Hello, World!!!" si no li passem cap paràmetre; si li passem un paràmetre anomenat name ens tornarà una salutació personalitzada canviant la paraula *World* pel nom especificat a name.

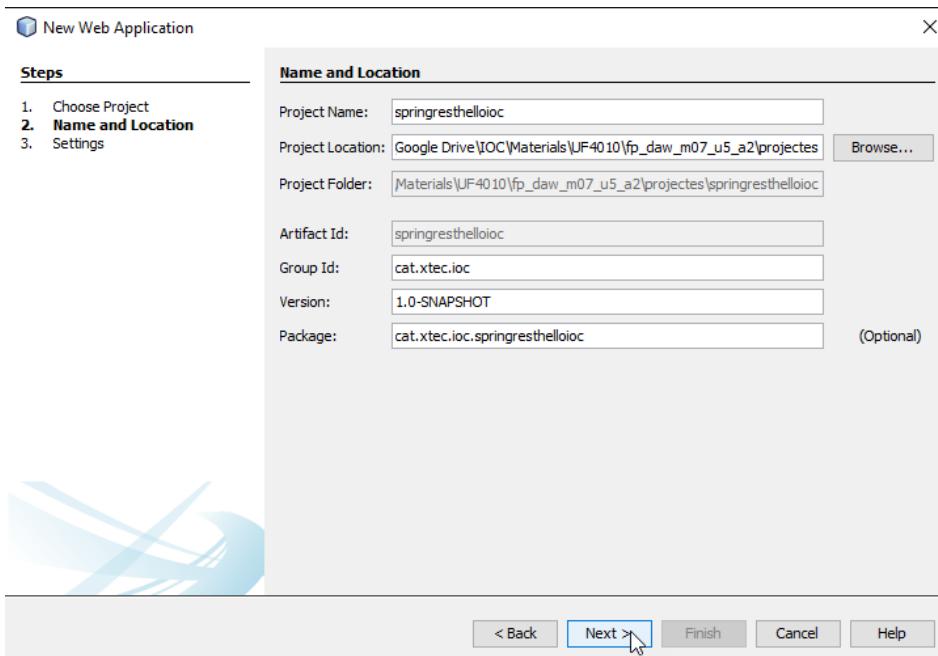
2.1.1 Creació i configuració inicial del projecte

Com que es tracta d'un exemple molt senzill no ens cal cap projecte de partida, simplement creeu a NetBeans un nou projecte Maven de tipus *Web Application*. Per fer-ho, feu *File / New Project* i us apareixerà l'assistent de creació de projectes. A l'assistent seleccioneu *Maven* i *Web Application*, tal com es veu en la figura 3.1.

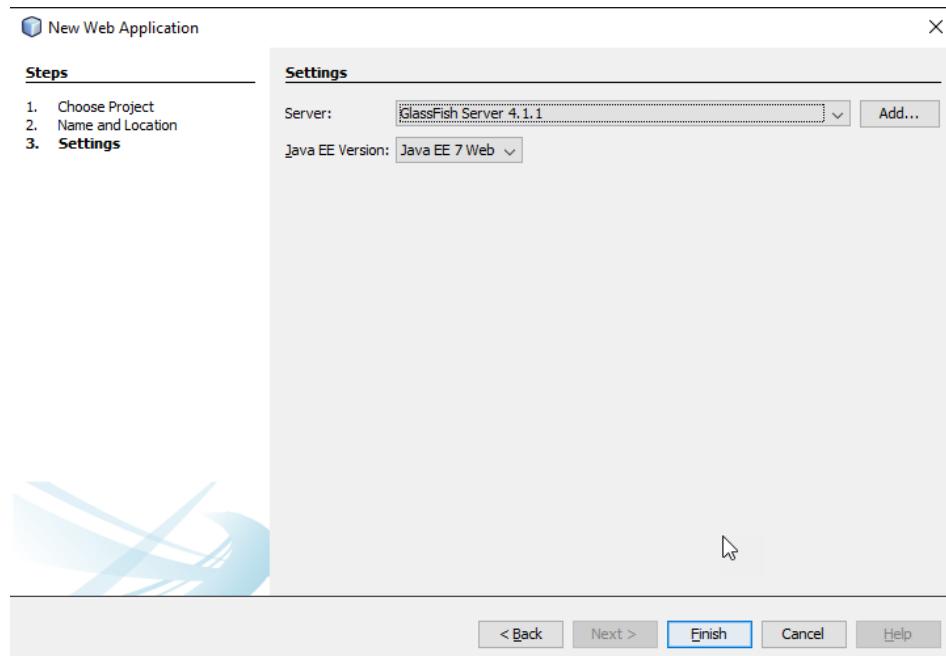
FIGURA 2.1. Creació de projectes a NetBeans

En la següent pantalla (vegeu la figura 3.2) triarem el nom del projecte (el podeu anomenar “Springresthelloioc”) i el paquet per defecte on anirà el codi font; per exemple, `cat.xtec.ioc.springresthelloioc`.

Hem triat per a l'exemple un projecte web amb Maven, però és perfectament vàlid fer-ho amb qualsevol altre tipus de projecte web.

FIGURA 2.2. Nom del nou projecte

En la següent pantalla (vegeu la figura 1.3) de l'assistent deixeu els valors per defecte i pulseu *Finish*.

FIGURA 2.3. Configuració del nou projecte

Un cop creat el projecte cal que modifiqueu el pom.xml per afegir les dependències cap a Spring; afegiu les següent línies al fitxer pom.xml:

```

1 <dependency>
2   <groupId>org.springframework</groupId>
3   <artifactId>spring-webmvc</artifactId>
4   <version>4.0.3.RELEASE</version>
5 </dependency>
```

També heu de crear el fitxer web.xml i configurar el *servlet DispatcherServlet* per tal que Spring MVC serveixi les peticions web; creeu el fitxer web.xml a la carpeta *Web Pages/WEB-INF* amb el següent contingut:

```

1 <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/
4     http://java.sun.com/xml/ns/javaee/web-app_3_0.
5     xsd">
6
6   <display-name>Spring RESTful Hello World</display-name>
7
8   <servlet>
9     <servlet-name>DispatcherServlet</servlet-name>
10    <servlet-class>org.springframework.web.servlet.DispatcherServlet</
11      servlet-class>
12    <init-param>
13      <param-name>contextConfigLocation</param-name>
14      <param-value>/WEB-INF/spring/DispatcherServlet-servlet.xml</param-
15        value>
16    </init-param>
17    <load-on-startup>1</load-on-startup>
18  </servlet>
19
20  <servlet-mapping>
21    <servlet-name>DispatcherServlet</servlet-name>
22    <url-pattern>/</url-pattern>
23  </servlet-mapping>
24
25</web-app>
```

Ens cal també crear el fitxer de configuració per a Spring; li hem indicat que el fitxer s'anomenarà DispatcherServlet-servlet.xml i serà a la ruta */WEB-INF/spring/*. Creeu primer la carpeta */WEB-INF/spring/* i després el fitxer DispatcherServlet-servlet.xml amb el següent contingut:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:context="http://www.springframework.org/schema/context"
5      xmlns:mvc="http://www.springframework.org/schema/mvc"
6      xsi:schemaLocation="http://www.springframework.org/schema/beans
7          http://www.springframework.org/schema/beans/spring-
8              beans.xsd
9          http://www.springframework.org/schema/context
10         http://www.springframework.org/schema/context/spring-
11             context-4.0.xsd
12         http://www.springframework.org/schema/mvc
13         http://www.springframework.org/schema/mvc/spring-mvc-
14             4.0.xsd">
15     <mvc:annotation-driven />
16     <context:component-scan base-package="cat.xtec.ioc" />
17
18     <bean class="org.springframework.web.servlet.view.
19         InternalResourceViewResolver">
20         <property name="prefix" value="/WEB-INF/views/" />
21         <property name="suffix" value=".jsp" />
22     </bean>
23
24 </beans>
```

Si recarregueu el pom.xml fent clic amb el botó dret damunt el nom del projecte i prement l'opció *Reload POM* del menú contextual ja tindreu el projecte configurat i llest per començar a crear el servei web RESTful amb Spring.

2.1.2 Creació del servei web RESTful

Ja heu creat el projecte que us servirà de base, ara cal que codifiqueu el servei web RESTful que ha de tornar la salutació “Hello, World!!!”; per fer-ho cal decidir primer dues coses: amb quin dels verbs HTTP ha de respondre el servei web i quin URI fareu servir per cridar-lo.

Aquest servei web ha d'**obtenir una representació del recurs** en format JSON, i per tant haurà de respondre el verb GET.

La decisió sobre quin URI utilitzar és força arbitrària; en aquest cas utilitzarem, per exemple, */hello*.

Per tant, a les peticions d'aquest tipus:

```
1  GET http://localhost:8080/resthelloioc/hello
```

ha de respondre amb:

```
1  {"id":1,"content":" Hello, World!!!"}
```

I a les peticions d'aquest tipus:

```
1 GET http://localhost:8080/resthelloioc/hello?name=User
```

ha de respondre amb:

```
1 {"id":1,"content":"Hello, User!"}
```

JSON

A l'exemple us demanem que treballeu amb una representació JSON. La simplicitat, lleugeresa i facilitat de lectura fan ideals aquesta representació per treballar amb aplicacions i dispositius que tenen restriccions pel que fa al volum de dades a intercanviar. Les aplicacions mòbils que consumeixin dades de serveis web RESTful són un molt bon exemple en aquest sentit; la quantitat d'informació que s'intercanviaran client i servidor per fer les operacions és molt menor en una aproximació JSON + RESTful que en una aproximació XML + SOAP.

El servei web tornarà la informació en format JSON. A la sortida, el camp *id* és un identificador únic per a la salutació, i el camp *content* representa la salutació.

Un cop decidit el funcionament del nostre servei, el següent serà implementar la classe que modelarà la representació de la salutació, és a dir, la classe que modelarà el que torna el servei web.

Creeu una classe Java anomenada *Greeting* al paquet `cat.xtec.ioc.springresthelloioc.domain` amb el següent codi:

```
1 public class Greeting {
2
3     private final long id;
4     private final String content;
5
6     public Greeting(long id, String content) {
7         this.id = id;
8         this.content = content;
9     }
10
11    public long getId() {
12        return id;
13    }
14
15    public String getContent() {
16        return content;
17    }
18 }
```

Spring fa servir la llibreria Jackson per fer les transformacions entre el format JSON i la seva representació Java, i a l'inrevés.

El següent pas és crear el controlador Spring, que s'encarregarà de servir les peticions; per fer-ho, creeu una classe Java anomenada *GreetingController* al paquet `cat.xtec.ioc.springresthelloioc.controller` amb el següent codi:

```
1 package cat.xtec.ioc.springresthelloioc.controller;
2
3 import cat.xtec.ioc.springresthelloioc.domain.Greeting;
4 import java.util.concurrent.atomic.AtomicLong;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.bind.annotation.RequestParam;
8 import org.springframework.web.bind.annotation.RestController;
9
10 @RestController
11 public class GreetingController {
12
13     private static final String template = "Hello, %s";
14     private final AtomicLong counter = new AtomicLong();
15
16     @RequestMapping(method = RequestMethod.GET, value = "/hello")
```

```

17  public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!"
18      ") String name) {
19      return new Greeting(counter.incrementAndGet(),
20                          String.format(template, name));
21  }

```

El codi del controlador és molt simple, però porta algunes particularitats que cal remarcar; fixeu-vos que:

- Hem anotat la classe amb `@RestController` per marcar la classe com un controlador REST on cada un dels seus mètodes tornarà un objecte del domini enlloc d'una vista. Aquesta és la diferència més gran entre un controlador MVC típic i un controlador per serveis web RESTful: el cos de la resposta HTTP es crea transformant un objecte del domini a JSON, i no es torna en format HTML.
- L'objecte `Greeting` que torna el mètode `greeting` cal que sigui transformat a JSON; Spring s'encarrega de fer això automàticament.
- Hem anotat el mètode `greeting` amb `@RequestMapping` indicant que les peticions a `/hello` les servirem amb aquest mètode.
- Hem anotat el mètode `greeting` amb `RequestMethod.GET` per indicar que respondrà a les peticions HTTP que es facin mitjançant el verb GET. Si no ho haguéssim fet, el mètode respondria a qualsevol dels verbs HTTP.
- Hem anotat el paràmetre del mètode `greeting` amb `@RequestParam` per indicar que el paràmetre `name` de la petició HTTP s'ha de vincular amb el paràmetre `name` del mètode. El paràmetre és opcional i, si no ve a la petició HTTP, farem servir “`World!!!!`” com a valor per defecte.
- En la implementació simplement creem i retornem l'objecte `Greeting`, que modela la salutació.
- Com que hi ha la llibreria Jackson al *classpath*, Spring tria la conversió de l'objecte `Greeting` a JSON per defecte.

I aquesta és tota la feina que cal fer per implementar el servei web, Spring farà la resta. Ara tan sols manca fer-ne el desplegament al servidor d'aplicacions i provar-lo.

2.1.3 Desplegament i prova del servei web RESTful

Un cop creat el servei web, cal que el desplegueu per tal de fer-lo accessible als clients i poder-lo provar. El procés de desplegament del servei web es fa desplegant l'aplicació Java EE que el conté mitjançant els mecanismes normals de desplegament de qualsevol aplicació Java EE.

Aquest procés és molt senzill, simplement cal que feu *Clean and Build* i després *Run* a NetBeans, i es farà el desplegament al servidor d'aplicacions que tingueu configurat per al projecte.

Si proveu ara accedint a l'URL localhost:8080/springresthelloioc/hello amb qualsevol navegador veureu que el servei web us torna la salutació en format JSON:

```
1 {"id":1,"content":"Hello, World!!!!"}

---


```

Proveu també especificant el paràmetre `name`, accedint a localhost:8080/springresthelloioc/hello?name=User, per exemple, i veureu que la salutació canvia:

```
1 {"id":2,"content":"Hello, User!"}

---


```

Amb això ja heu creat, configurat, desplegat i provat un servei web RESTful molt senzill amb Spring.

2.2 Testejant el servei web ""Hello, World!!!!"

Aprofitarem les capacitats que proporciona Spring, i més concretament Spring MVC, per crear un conjunt de tests que van a cavall entre un test unitari i un test d'integració. Aquests tests us permetran provar els serveis web sense necessitat de tenir-los desplegats i executant-se a un servidor d'aplicacions. Testejarem amb Spring el típic exemple de "*Hello, World!!!!*".

2.2.1 Creació i configuració inicial del projecte

Descarregueu el codi del projecte "Springtestresthelloioc" en l'estat inicial d'aquest apartat en l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

El projecte té un servei web RESTful configurat a la classe `GreetingController` amb el següent codi:

```
1 package cat.xtec.ioc.springresthelloioc.controller;
2
3 import cat.xtec.ioc.springresthelloioc.domain.Greeting;
4 import java.util.concurrent.atomic.AtomicLong;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.bind.annotation.RequestParam;
8 import org.springframework.web.bind.annotation.RestController;
9
10 @RestController
11 public class GreetingController {
12
13     private static final String template = "Hello, %s";
14     private final AtomicLong counter = new AtomicLong();
15
16     @RequestMapping(method = RequestMethod.GET, value = "/hello")
17     public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!"))
18         String name) {
19             return new Greeting(counter.incrementAndGet(),

---


```

```

19         String.format(template, name));
20     }
21 }
```

Aquest servei web és un servei web RESTful desenvolupat amb Spring que torna “Hello, World!!!” si no li passem cap paràmetre, i si li passem un paràmetre anomenat `name` torna una salutació personalitzada canviant la paraula *World* pel nom especificat a `name`.

El servei web torna la salutació en format JSON i respon a les peticions GET a `/hello` amb:

```
1 {"id":1,"content":" Hello, World!!!"} 
```

I a les peticions GET a `/hello?name=User` amb:

```
1 {"id":1,"content":"Hello, User!"} 
```

El test que escriurem utilitzarà les capacitats que té Spring MVC per fer test unitari dels *endpoints* configurats a un controlador RESTful. La principal diferència entre aquest tipus de tests i els tests d’integració és que en aquest cas **no ens cal tenir el servei web que volem provar desplegat i executant-se a un servidor d’aplicacions**.

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

Els **tests d’integració** difereixen dels tests unitaris, ja que no testegen el codi de forma aïllada, sinó que requereixen que el codi a testejar estigui desplegat al servidor d’aplicacions per funcionar.

Els tests unitaris amb el bastiment proporcionat per Spring MVC es basen en JUnit 4. JUnit és un *framework* de test que utilitza anotacions per identificar els mètodes que especificuen un test. A JUnit, un test, ja sigui unitari o d’integració, és un mètode que s’especifica en una classe que només s’utilitza per al test. Això s’anomena una *classe de test*. Un mètode de test amb JUnit 4 es defineix amb l’anotació `@org.junit.Test`. En aquest mètode s’utilitza un mètode d’asserció en el qual es comprova el resultat esperat de l’execució de codi en comparació del resultat real.

2.2.2 Creació i execució dels tests unitaris

Ara toca començar a crear els tests d’integració per provar el servei web. Per fer-ho, creeu un nou paquet dins de *Test Packages* anomenat, per exemple, `cat.xtec.ioc.springresthelloioc.controller`, i una classe Java anomenada `GreetingControllerTest`:

```

1 package cat.xtec.ioc.springresthelloioc.controller;
2
3 import org.junit.Before;
4 import org.junit.Test;
```

```
5  import org.junit.runner.RunWith;
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.http.MediaType;
8  import org.springframework.http.converter.HttpMessageConverter;
9  import org.springframework.http.converter.json.
   MappingJackson2HttpMessageConverter;
10 import org.springframework.mock.http.MockHttpOutputMessage;
11 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
12 import org.springframework.test.context.web.WebAppConfiguration;
13 import org.springframework.test.web.servlet.MockMvc;
14 import org.springframework.web.context.WebApplicationContext;
15
16 import java.io.IOException;
17 import java.nio.charset.Charset;
18 import java.util.Arrays;
19 import static junit.framework.Assert.assertNotNull;
20 import static org.hamcrest.CoreMatchers.is;
21
22 import org.springframework.boot.test.SpringApplicationConfiguration;
23 import static org.springframework.test.web.servlet.request.
   MockMvcRequestBuilders.*;
24 import static org.springframework.test.web.servlet.result.MockMvcResultMatchers
   ..*;
25 import static org.springframework.test.web.servlet.setup.MockMvcBuilders.*;
26
27
28 @RunWith(SpringJUnit4ClassRunner.class)
29 @SpringApplicationConfiguration(classes = Application.class)
30 @WebAppConfiguration
31 public class GreetingControllerTest {
32     private MediaType contentType = new MediaType(MediaType.APPLICATION_JSON.
   getTypeId(),
33           MediaType.APPLICATION_JSON.getSubtype(),
34           Charset.forName("utf8"));
35
36     private MockMvc mockMvc;
37
38     private HttpMessageConverter mappingJackson2HttpMessageConverter;
39
40     @Autowired
41     private WebApplicationContext webApplicationContext;
42
43     @Autowired
44     void setConverters(HttpMessageConverter<?>[] converters) {
45         this.mappingJackson2HttpMessageConverter = Arrays.asList(converters).
   stream()
46             .filter(hmc -> hmc instanceof MappingJackson2HttpMessageConverter)
47             .findAny()
48             .orElse(null);
49
50         assertNotNull("the JSON message converter must not be null",
51             this.mappingJackson2HttpMessageConverter);
52     }
53
54     @Before
55     public void setup() throws Exception {
56         this.mockMvc = webAppContextSetup(webApplicationContext).build();
57     }
58
59
60     protected String json(Object o) throws IOException {
61         MockHttpOutputMessage mockHttpOutputMessage = new MockHttpOutputMessage
   ();
62         this.mappingJackson2HttpMessageConverter.write(
63             o, MediaType.APPLICATION_JSON, mockHttpOutputMessage);
64         return mockHttpOutputMessage.getBodyAsString();
65     }
66 }
67 }
```

Fixeu-vos que:

- Anotem la classe amb `@RunWith(SpringJUnit4ClassRunner.class)` per indicar que es tracta d'un test unitari que s'executarà amb l'executor de JUnit que porta Spring.
- Anotem la classe amb `@SpringApplicationConfiguration(classes = Application.class)` per indicar que crearem una classe que carregui la configuració Spring. Aquesta és una de les diverses maneres de fer-ho, també es podria carregar la configuració amb fitxers XML.
- Anotem la classe amb `@WebAppConfiguration` per indicar a JUnit que es tracta d'un test unitari per a Spring MVC i que s'ha d'executar amb un context d'aplicació web i no d'aplicació *stand-alone*.
- Anotem un mètode anomenat `setup` amb `@Before` per indicar a JUnit que aquest mètode s'executi cada cop que es creï la classe de test. En aquest mètode creem un objecte de tipus `MockMvc` que serà el que ens proporcionarà tota la infraestructura necessària per fer els tests sense haver de desplegar l'aplicació a cap servidor d'aplicacions.
- El mètode `json` és un mètode que ens torna la representació en format JSON d'un objecte.
- S'han configurat tots els conversors que es requereixen per parsejar les peticions i les respostes.

Per tal que Spring pugui carregar la configuració cal una classe anomenada `Application` al paquet `cat.xtec.ioc.springresthelloioc.controller` amb el següent codi:

```

1 package cat.xtec.ioc.springresthelloioc.controller;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class Application {
8
9     public static void main(String[] args) {
10         SpringApplication.run(Application.class, args);
11     }
12 }
```

Aquesta classe ja la teniu creada al projecte de partida.

El primer test que farem serà un test que **comprovi que la petició `/hello` sense cap paràmetre torna la representació JSON de la salutació “Hello, World!!!”**; per fer-ho, creeu un mètode anomenat `greetingShouldReturnHelloWorldWithoutName` dins la classe `GreetingControllerTest` amb el següent codi:

```

1 @Test
2 public void greetingShouldReturnHelloWorldWithoutName() throws Exception {
3     mockMvc.perform(get("/hello"))
```

```

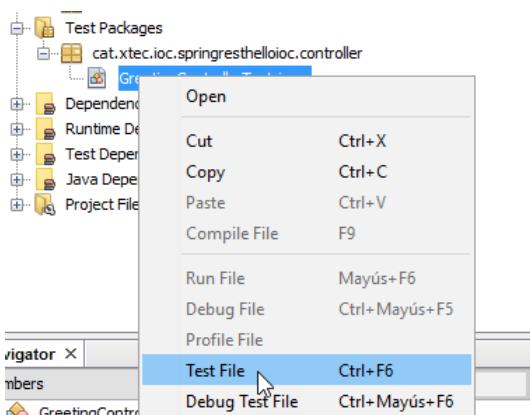
4     .andExpect(status().isOk())
5     .andExpect(content().contentType(contentType))
6     .andExpect(jsonPath("$.content", is("Hello, World!!!")));
7 }
```

Si analitzem el codi veiem diverses coses importants: la primera és que hem anotat el mètode `greetingShouldReturnHelloWorldWithoutName` amb l'anotació `@Test` per indicar que es tracta d'un mètode de test.

També que fa servir l'objecte MockMVC que heu creat per fer una petició de tipus GET a l' URI `/hello`; comproveu que el resultat de la petició sigui un 200 (OK), que el *content type* sigui JSON i que tingui al camp *content* la salutació per defecte `"Hello, World!!!"`.

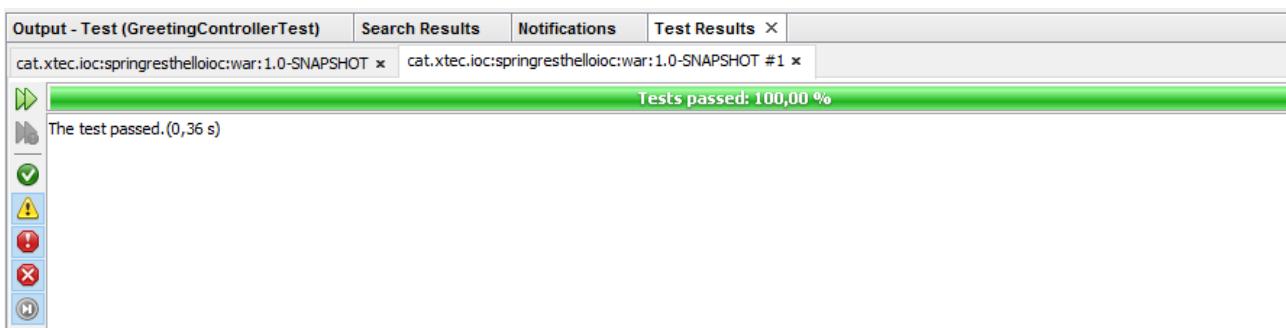
Ja podem executar el test que hem creat fent *Test File* (vegeu la figura 2.4) al menú contextual de la classe de test.

FIGURA 2.4. Execució del test

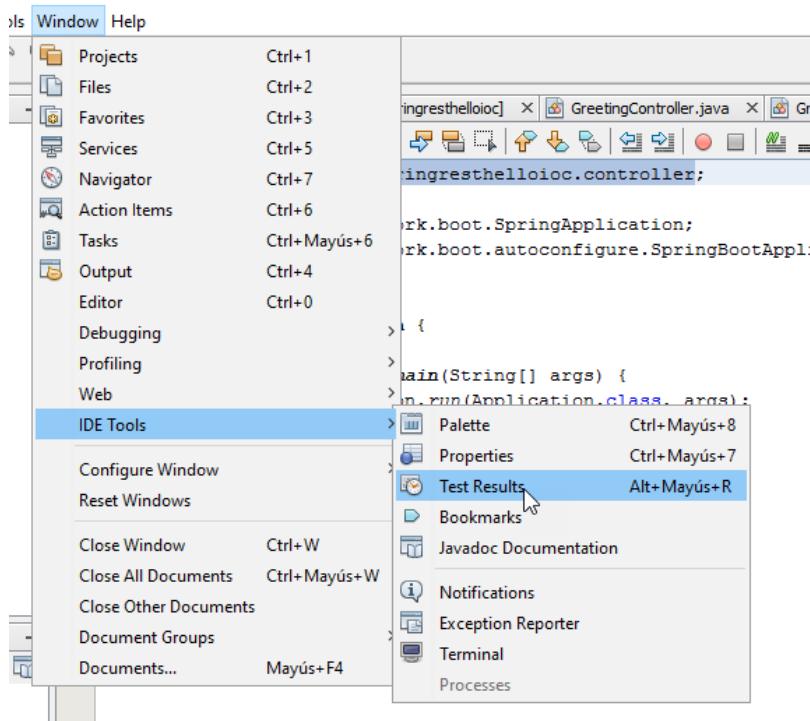


Si tot ha anat bé veureu el resultat a la finestra de *Test* (vegeu la figura 2.5).

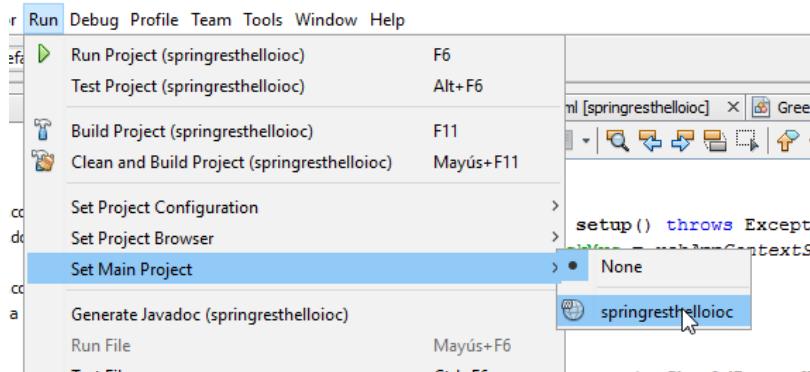
FIGURA 2.5. Resultat de l'execució del test



Tot i que el resultat de l'execució dels test es pot veure a la finestra de sortida de NetBeans, és molt més còmode visualitzar-ho a la finestra de resultats de tests que proporciona també NetBeans; per mostrar aquesta finestra aneu al menú *Window / IDE Tools / Test Results* (vegeu la figura 2.6).

FIGURA 2.6. Finestra de resultats dels tests

A Netbeans, els tests es poden executar de forma individual, és a dir, classe per classe o tots els del projecte. Si voleu executar tots els tests d'un projecte, primer heu de designar com a projecte principal el projecte "Springteststhelloioc", tal com podeu veure en la figura 2.7.

FIGURA 2.7. Assignació del projecte com a projecte principal

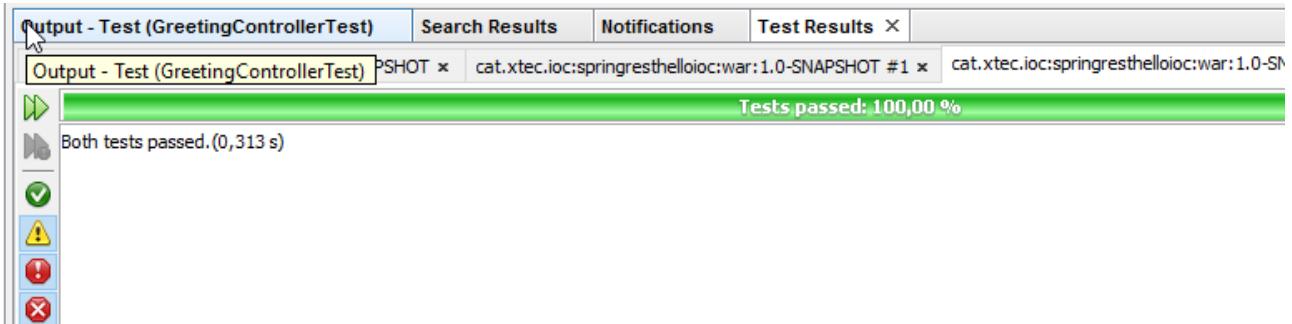
El segon test que farem serà un test que **comprovi que la petició `/hello?name=User` torna la representació JSON de la salutació “Hello, User”**; per fer-ho, creeu un mètode anomenat `greetingShouldReturnHelloNameWith Name` dins la classe `GreetingControllerTest` amb el següent codi:

```

1 @Test
2 public void greetingShouldReturnHelloNameWith Name() throws Exception {
3     mockMvc.perform(get("/hello?name=User"))
4         .andExpect(status().isOk())
5         .andExpect(content().contentType(contentType))
6         .andExpect(jsonPath("$.content", is("Hello, User")));
7 }
```

El codi és molt similar al primer test; executeu-lo de la mateixa manera i comproveu que el test s'executa correctament (vegeu la figura 2.8).

FIGURA 2.8. Resultat de l'execució dels dos tests



Aquest dos exemples us donen la base per tal que pugueu fer tots els tests unitaris que se us acudeixin per als serveis web RESTful que desenvolupeu amb Spring i així tenir-los totalment provats.

2.3 El servei web de gestió d'equips de futbol. Operacions CRUD

Veurem els conceptes referents a la creació de serveis web RESTful amb Spring desenvolupant un servei web RESTful que permeti fer operacions sobre equips de futbol i els jugadors que els integren. Farem les operacions típiques CRUD amb els jugadors, una operació que mostra el llistat d'equips, una que mostra els jugadors d'un equip i una altra que mostra les dades d'un equip.

CRUD és l'acrònim en anglès de les operacions de creació (Create), lectura (Read), actualització (Update) i esborrat (Delete).

Concretament, el servei web que farem tindrà les següents operacions:

- llistar tots els equips
- consulta de les dades d'un equip
- llistar tots els jugadors d'un equip
- consulta d'un jugador d'un equip (operació Read CRUD)
- creació d'un jugador en un equip (operació Create CRUD)
- actualització de les dades d'un jugador d'un equip (operació Update CRUD)
- esborrar un jugador d'un equip (operació Delete CRUD)

Per fer-ho emprareu una aplicació web ja desenvolupada que segueix una arquitectura per capes i hi afegireu una capa de serveis on creareu i publicareu el servei web de gestió d'equips catàleg com a servei web RESTful. La representació dels recursos que fareu servir en tot l'exemple serà JSON.

2.3.1 Creació i configuració inicial del projecte

L'aplicació de la qual partireu s'anomena “Springrestteamsioc” i servirà per veure com podeu exposar algunes funcionalitats de la capa de serveis d'una aplicació mitjançant serveis web RESTful. Es tracta d'un projecte Spring MVC senzill que segueix una arquitectura típica per capes per tal d'aconseguir una alta reusabilitat, un baix acoblament i una alta cohesió a l'aplicació.

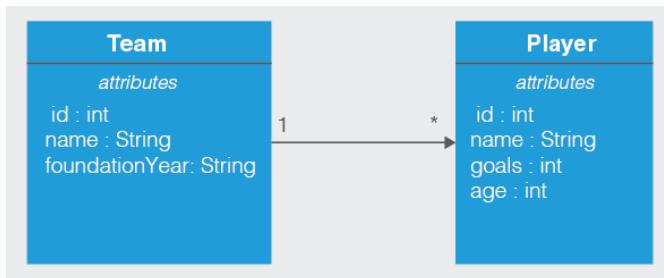
El projecte consta de quatre capes:

- capa de presentació
- capa de domini
- capa de serveis
- capa de persistència

Com que l'aplicació d'exemple no ha de proporcionar interfície gràfica, fareu servir la capa de presentació com la capa on publicareu els serveis RESTful; seria igualment vàlid fer-ho a una capa de serveis independent.

El model de domini ja el teniu implementat i és molt senzill: hi ha una entitat Team que representa un equip i una entitat Player que representa els jugadors d'un equip. Entre Team i Player hi ha una associació $1:N$ per indicar que un jugador pertany a un equip i que un equip està format per N jugadors (vegeu la figura 2.9).

FIGURA 2.9. Entitats Team i Player



La representació en JSON d'un equip és la següent:

```

1 {
2   "id":1,
3   "name":"F.C. Barcelona",
4   "foundationYear":"1899"
5 }
```

I la representació JSON d'un jugador és la següent:

```

1 {
2   "id":1,
3   "name":"Lionel Messi",
```

Repositoris 'in memory'

Tot i que podríem haver optat per fer l'exemple amb un repositori connectat a una font de dades persistent com una base de dades relacional i utilitzar l'API JPA per definir les entitats del projecte, s'ha considerat que això afegeix “soroll” a l'exemple i us faria fer algunes tasques de configuració que no són pròpies de l'objectiu principal. Per aquest motiu fareu servir repositoris *in memory* tant per als equips com per als jugadors.

```

4   "goals":472,
5   "age":29,
6   "teamId":1
7 }
```

La capa de serveis serà la que treballarem i haurà de proporcionar **un servei web RESTful** que permeti als clients fer les següents operacions sobre els equips:

- llistar tots els equips
- consulta de les dades d'un equip
- llistar tots els jugadors d'un equip
- consulta d'un jugador d'un equip
- creació d'un jugador en un equip
- actualització de les dades d'un jugador d'un equip
- esborrar un jugador d'un equip

La capa de persistència també la teniu implementada i conté dos objectes repositori que permeten mapar les dades de les fonts de dades amb els objectes del domini. En el nostre cas, per simplicitat, farem servir repositoris *in memory* que tindran una llista precarregada amb els equips i els jugadors de cada equip.

2.3.2 Creació i prova del servei web RESTful

Descarregueu el codi del projecte "Springteamsioc" de l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

Un cop fet, creareu una classe que exposarà les operacions que voleu realitzar sobre els equips; anomenareu la classe **TeamsController**, la creareu al paquet **cat.xtec.ioc.controller** i l'anotareu amb **@RestController**:

```

1 package cat.xtec.ioc.controller;
2
3 import org.springframework.web.bind.annotation.RestController;
4
5 @RestController
6 public class TeamsController {
7 }
```

Simplement heu anotat la classe amb **@RestController** per marcar la classe com un controlador REST on cada un dels seus mètodes tornarà un objecte del domini enlloc d'una vista. Aquesta és la diferència més gran entre un controlador MVC típic i un controlador per a serveis web RESTful: el cos de la resposta HTTP es crea transformant un objecte del domini a JSON i no es torna en format HTML.

Un cop creada la classe cal que hi afegiu una referència al repositori dels equips fent que Spring el carregui amb el seu mòdul d'injecció de dependències:

```

1 @Autowired
2 private TeamRepository teamRepository;
3
4 public TeamsController() {
5 }
6
7 public TeamsController(TeamRepository teamRepository) {
8     this.teamRepository = teamRepository;
9 }
```

I ja podeu començar a codificar el primer servei que voleu oferir; per exemple, la consulta de tots els equips. Per fer-ho, creeu un mètode anomenat getAll amb el següent codi:

```

1 @RequestMapping(method = RequestMethod.GET, value = "/teams")
2 public @ResponseBody List<Team> getAll() {
3     return this.teamRepository.getAll();
4 }
```

Hem anotat el mètode amb l'anotació `@RequestMapping(method = RequestMethod.GET, value = '/teams')` per indicar que aquest mètode respondrà a peticions HTTP de tipus GET a l'URL `/teams`.

Hem anotat el mètode amb `@ResponseBody` per indicar a Spring que torni el resultat en un format que sigui entenedor per al client a partir de les capçaleres HTTP que envia amb la petició. Per defecte, es torna en format JSON.

Tornem la llista d'equips com una llista d'objectes Java de tipus `Team` i Spring s'encarregarà de transformar-los a format JSON.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL localhost:8080/springrestteamsioc/teams; si tot ha anat bé, veureu la representació JSON dels equips al navegador:

```

1 [
2 {"id":1,"name":"F.C.Barcelona","foundationYear":"1899"},
3 {"id":2,"name":"Real Madrid","foundationYear":"1902"}
4 ]
```

Passem a crear ara el servei de consulta de les dades d'un equip; per fer-ho, creeu un mètode anomenat getById amb el següent codi:

```

1 @RequestMapping(value = "/teams/{id}", method = RequestMethod.GET)
2 public @ResponseBody Team getById(@PathVariable int id) {
3     return this.teamRepository.get(id);
4 }
```

El codi és molt similar a la consulta d'equips, però com a particularitat cal comentar que ara tornem un objecte Java de tipus `Team` que Spring convertirà a JSON abans de tornar-lo al client i que hem anotat el mètode amb l'anotació `@RequestMapping(value = '/teams/{id}', method = RequestMethod.GET)` per indicar que la informació de l'equip la tornarem quan ens arribin peticions GET a l'URI `/teams/{id}`; és a dir, el que posem darrere de `/teams` serà l'identificador de l'equip que volem consultar.

Quan NetBeans us demani quins imports voleu afegir específicau, a la major part de casos, els del paquet `org.springframework.web.bind.annotation`.

El paràmetre que rep el mètode s'ha anotat amb l'anotació `@PathVariable int id`. Aquesta és la forma que proporciona Spring per extreure els paràmetres d'una petició. En aquest cas estem extraient un paràmetre del *path* i el passem com a *int* al mètode `getById`.

Spring MVC proporciona un ampli conjunt d'anotacions per fer aquesta tasca fàcil, entre elles `@PathParam`, `@RequestParam`, `@CookieValue`, `@RequestHeader`, etc.

Per exemple, podríem utilitzar `@RequestParam("id")` per extreure un paràmetre que vingués en una petició d'aquesta forma: <localhost:8080/springrestteamsioc/teams?id=2>.

Per provar si funciona desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL <localhost:8080/springrestteamsioc/teams/1>; si tot ha anat bé, veureu la representació JSON de l'equip consultat al navegador:

```

1  [
2    {
3      "id":1,
4      "name":"F.C. Barcelona",
5      "foundationYear":"1899"
6    }
7  ]

```

Ja heu creat les dues operacions que volíeu per als equips, ara començareu amb les operacions que voleu oferir per als jugadors d'un equip. Aquestes operacions les podríeu posar el mateix controlador on heu posat les operacions per als equips, però per fer un codi més net i que respecti al màxim el principi de disseny SRP (de l'anglès *Single Responsibility Principle*) ho fareu en un controlador diferent.

Creeu una classe que exposarà les operacions que voleu realitzar sobre els jugadors dels equips; anomenareu la classe `PlayerController`, la creareu al paquet `cat.xtec.ioc.controller` i l'anotareu amb `@RestController`:

```

1 package cat.xtec.ioc.controller;
2
3 import org.springframework.web.bind.annotation.RestController;
4 import org.springframework.web.bind.annotation.RequestMapping;
5
6 @RestController
7 @RequestMapping("/teams/{teamId}/players")
8 public class PlayerController {
9 }

```

Simplement heu anotat la classe amb `@RestController` per marcar la classe com un controlador REST on cada un dels seus mètodes tornarà un objecte del domini enlloc d'una vista.

També heu anotat la classe amb `@RequestMapping("/teams/{teamId}/players")` per indicar que tots els mètodes del controlador s'han d'accendir amb aquest format. Amb això, tots els mètodes del controlador podran extreure del *path* l'identificador de l'equip i no us caldrà repetir-ho a cada un d'ells.

Un cop creada la classe, cal que hi afegiu una referència al repositori dels jugadors dels equips fent que Spring el carregui amb el seu mòdul d'injecció de dependències:

```

1 @Autowired
2 private PlayerRepository playerRepository;
3
4 public PlayerController() {
5 }
6
7 public PlayerController(PlayerRepository playerRepository) {
8     this.playerRepository = playerRepository;
9 }
```

Creeu primer la funcionalitat que permet la consulta de tots els jugadors d'un equip. Per fer-ho, creeu un mètode anomenat `getTeamPlayers` amb el següent codi:

```

1 @RequestMapping(method = RequestMethod.GET)
2 public @ResponseBody List<Player> getTeamPlayers(@PathVariable int teamId) {
3     return this.playerRepository.getByTeam(teamId);
4 }
```

Heu anotat el mètode amb l'anotació `@RequestMapping(method = RequestMethod.GET)` per indicar que aquest mètode respondrà a peticions HTTP de tipus GET i ho farà a l'URI per defecte que heu configurat al controlador (`/teams/{teamId}/players`).

Heu anotat el mètode amb `@ResponseBody` per indicar a Spring que torni el resultat en un format que sigui comprensible per al client a partir de les capçaleres HTTP que el client envia amb la petició. Per defecte, es torna en format JSON.

Torneu la llista d'equips com una llista d'objectes Java de tipus `Player` i Spring s'encarregarà de transformar-los a format JSON.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL localhost:8080/springrestteamsioc/teams/1/players; si tot ha anat bé, veureu la representació JSON dels jugadors del FC Barcelona al navegador:

```

1 [
2 {"id":1,"name":"Lionel Messi","goals":472,"age":29,"teamId":1},
3 {"id":2,"name":"Luis Suarez","goals":100,"age":29,"teamId":1}
4 ]
```

Passareu a crear ara el servei de consulta de les dades d'un jugador d'un equip (l'operació Read de CRUD); per fer-ho, creeu un mètode anomenat `getById` amb el següent codi:

```

1 @RequestMapping(value = "{playerId}", method = RequestMethod.GET)
2 public @ResponseBody Player getById(@PathVariable int teamId, @PathVariable int
3     playerId) {
4     return this.playerRepository.get(teamId, playerId);
5 }
```

El codi és molt similar a la consulta dels jugadors d'un equip; com a particularitat, cal comentar que ara tornem un objecte Java de tipus `Player` que Spring convertirà a JSON abans de tornar-lo al client i que hem anotat el mètode amb l'anotació `@RequestMapping(value = "{playerId}", method`

= RequestMethod.GET) per indicar que la informació del jugador la tornarem quan ens arribin peticions GET a l'URI `/teams/{teamId}/players/{playerId}`; és a dir, el que posem darrere de `/teams` serà l'identificador de l'equip al qual pertany el jugador, i el que posem darrere de `players` serà l'identificador del jugador que volem consultar.

Els paràmetres que rep el mètode s'han anotat amb les anotacions @PathVariable int `teamId` i @PathVariable int `playerId`, respectivament. Aquesta és la forma que proporciona Spring per extreure els paràmetres d'una petició. En aquest cas, estem extraient dos paràmetres del *path* i els passem com a *int* al mètode `getById`.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL localhost:8080/springrestteamsioc/teams/1/players/1; si tot ha anat bé, veureu la representació JSON del jugador “Lionel Messi” del “FC Barcelona” al navegador:

```

1  [
2    {
3      "id":1,
4      "name":"Lionel Messi",
5      "goals":472,
6      "age":29,
7      "teamId":1
8    }
]
```

Un cop fetes les dues operacions de consulta sobre els jugadors d'un equip passareu ara a crear el servei que permeti donar d'alta un jugador d'un equip (l'operació Create de CRUD); per fer-ho, creeu un mètode anomenat `create` amb el següent codi:

```

1  @RequestMapping(method = RequestMethod.POST)
2  public @ResponseBody ResponseEntity<Player> create(@PathVariable int teamId,
3      @RequestBody Player player) {
4      player.setTeamId(teamId);
5      this.playerRepository.add(player);
6      return new ResponseEntity<>(player, HttpStatus.CREATED);
7  }
```

Heu anotat el mètode amb l'anotació @RequestMapping(method = RequestMethod.POST) per indicar que aquest mètode respondrà a peticions HTTP de tipus POST i ho farà a l'URI per defecte que heu configurat al controlador (`/teams/{teamId}/players`).

La resposta que tornem és de tipus `ResponseEntity<Player>` per tal que hi puguem afegir la representació del jugador creat i el codi HTTP (en aquest cas un codi 201, CREATED, per indicar que el recurs s'ha creat satisfactoriament).

Fixeu-vos que tan sols amb aquesta informació Spring és capaç, quan rep una petició POST a l'URI `/teams/{teamId}/players` amb una representació JSON d'un jugador, de crear un objecte de tipus `Player` i passar-lo al mètode `create`.

Ara tocaria provar aquesta funcionalitat, però tenim un problema: com podem enviar peticions POST sense fer una aplicació web amb un formulari? Per fer una petició GET tan sols hem de posar l'URL al navegador i ja ho tenim, però

Per documentar i provar API RESTful teniu moltes alternatives, entre les quals cal destacar Swagger swagger.io i Postman www.getpostman.com, que és una extensió per al navegador Google Chrome.

per fer peticions POST caldrà alguna utilitat extra. La nostra proposta és que feu servir **cURL**, que és una eina molt útil per fer peticions HTTP de diferents tipus i és multiplataforma. Si feu servir Linux possiblement ja la tingueu instal·lada al sistema; en cas que utilitzeu Windows, la podeu descarregar en el següent enllaç: curl.haxx.se/download.html.

La sintaxi de cURL per fer peticions és molt senzilla; per exemple, per consultar tots els equips feu un *command prompt*:

```
1 curl localhost:8080/springrestteamsioc/teams
```

Afegeix el paràmetre `-v` (`verbose`) a les crides a cURL si voleu veure més informació sobre les peticions i respostes que feu.

Per provar la funcionalitat que permet afegir un jugador al FC Barcelona desplegueu el projecte fent *Run* a NetBeans i feu una petició POST a l'URL localhost:8080/springrestteamsioc/teams/1/players especificant la informació del jugador amb JSON; això ho podeu fer amb cURL i la següent comanda:

```
1 curl -H "Content-Type: application/json" -X POST -d "{\"id\":\"3\", \"name\":\"Neymar da Silva Santos\", \"age\":24, \"goals\":100}" http://localhost:8080/springrestteamsioc/teams/1/players
```

Fixeu-vos: li diem que farem una petició POST amb el paràmetre `-X`, li especificarem el format JSON del jugador amb el paràmetre `-d` i li indiquem que el format és JSON especificant-ho a la capçalera amb el paràmetre `-H`.

Executeu la comanda anterior i després consulteu els jugadors del FC Barcelona amb:

```
1 curl localhost:8080/springrestteamsioc/teams/1/players
```

Si tot ha anat bé, veureu que Neymar s'ha afegit com a jugador del FC Barcelona:

```
1 [
2   {"id":1,"name":"Lionel Messi","goals":472,"age":29,"teamId":1},
3   {"id":2,"name":"Luis Suarez","goals":100,"age":29,"teamId":1},
4   {"id":3,"name":"Neymar da Silva Santos","goals":100,"age":24,"teamId":1}
5 ]
```

Creem ara el servei que permeti modificar la informació d'un jugador (l'operació Update de CRUD); per fer-ho, creeu un mètode anomenat update amb el següent codi:

```
1 @RequestMapping(method = RequestMethod.PUT)
2 public @ResponseBody ResponseEntity<Player> update(@PathVariable int teamId,
3   @RequestBody Player player) {
4     player.setTeamId(teamId);
5     this.playerRepository.update(player);
6     return new ResponseEntity<>(player, HttpStatus.OK);
}
```

Les consideracions en aquest cas són les mateixes que hem vist per al cas de la creació d'un jugador, l'únic que canvia és que farem servir el verb PUT en lloc del verb POST.

Per provar la funcionalitat que permet modificar la informació d'un jugador desplegueu el projecte fent *Run* a NetBeans i feu una petició PUT a l'URL localhost:8080/springrestteamsioc/teams/1/players

calhost:8080/springrestteamsioc/teams/1/players especificant la nova informació del jugador; això ho podeu fer amb cURL i la següent comanda:

```
1 curl -H "Content-Type: application/json" -X PUT -d "{\"id\":\"3\",\"name\":\"Neymar da Silva Santos\",\"age\":24,\"goals\":120}" http://localhost:8080/springrestteamsioc/teams/1/players
```

Pèrdua de dades

Us pot passar que en afegir el mètode `update` per modificar la informació d'un jugador es torni a desplegar l'aplicació, i, com que feu servir un repositori *in memory*, es perdin les dades dels jugadors que heu donat d'alta. Si us trobeu en aquest cas simplement cal que torneu a fer l'alta del jugador *Neymar* i després en feu la modificació.

Executeu la comanda anterior i després consulteu la informació del jugador que heu modificat amb:

```
1 curl http://localhost:8080/springrestteamsioc/teams/1/players/3
```

Si tot ha anat bé, veureu que el nombre de gols que ha marcat Neymar ha passat de 100 a 120:

```
1 [
2 {"id":3,"name":"Neymar da Silva Santos","goals":120,"age":24,"teamId":1}
3 ]
```

Creem ara el servei que permet esborrar un jugador d'un equip (l'operació `Delete` de CRUD); per fer-ho, creeu un mètode anomenat `delete` amb el següent codi:

```
1 @RequestMapping(value = "{playerId}", method = RequestMethod.DELETE)
2 public @ResponseBody ResponseEntity<Player> delete(@PathVariable int teamId,
3                                                 @PathVariable int playerId) {
4     this.playerRepository.delete(teamId, playerId);
5     return new ResponseEntity<>(HttpStatus.OK);
}
```

Les consideracions en aquest cas són les mateixes que hem vist per al cas de la consulta d'un jugador d'un equip, i l'únic que canvia és que fareu servir el verb `DELETE` en lloc del verb `GET`.

Per provar la funcionalitat que permet esborrar un jugador d'un equip desplegueu el projecte fent *Run* a NetBeans i feu una petició `DELETE` a l'URL localhost:8080/springrestteamsioc/{teamId}/players/{playerId} especificant l'equip al qual pertany el jugador i l'identificador del jugador que volem esborrar; això ho podeu fer amb cURL i la següent comanda (per exemple, per esborrar el jugador Neymar del FC Barcelona):

```
1 curl -X DELETE localhost:8080/springrestteamsioc/teams/1/players/3
```

Executeu la comanda anterior i després consulteu els jugadors del FC Barcelona amb:

```
1 curl localhost:8080/springrestteamsioc/teams/1/players/
```

Si tot ha anat bé, veureu que Neymar ja no és jugador del FC Barcelona:

```
1 [
2 {"id":1,"name":"Lionel Messi","goals":472,"age":29,"teamId":1},
3 {"id":2,"name":"Luis Suarez","goals":100,"age":29,"teamId":1}
4 ]
```

I amb això ja heu codificat i provat el servei web RESTful que us permet treballar amb els equips i els seus jugadors. El codi final de la classe TeamController és el següent:

```

1 package cat.xtec.ioc.controller;
2
3 import cat.xtec.ioc.domain.Team;
4 import cat.xtec.ioc.domain.repository.TeamRepository;
5 import java.util.List;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.web.bind.annotation.PathVariable;
8 import org.springframework.web.bind.annotation.RequestMapping;
9 import org.springframework.web.bind.annotation.RequestMethod;
10 import org.springframework.web.bind.annotation.ResponseBody;
11 import org.springframework.web.bind.annotation.RestController;
12
13 @RestController
14 public class TeamsController {
15
16     @Autowired
17     private TeamRepository teamRepository;
18
19     public TeamsController() {
20     }
21
22     public TeamsController(TeamRepository teamRepository) {
23         this.teamRepository = teamRepository;
24     }
25
26     @RequestMapping(method = RequestMethod.GET, value = "/teams")
27     public @ResponseBody
28     List<Team> getAll() {
29         return this.teamRepository.getAll();
30     }
31
32     @RequestMapping(value = "/teams/{id}", method = RequestMethod.GET)
33     public @ResponseBody
34     Team getById(@PathVariable int id) {
35         return this.teamRepository.get(id);
36     }
37 }
```

I el de la classe PlayerController és el següent:

```

1 package cat.xtec.ioc.controller;
2
3 import cat.xtec.ioc.domain.Player;
4 import cat.xtec.ioc.domain.repository.PlayerRepository;
5 import java.util.List;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.http.HttpStatus;
8 import org.springframework.http.ResponseEntity;
9 import org.springframework.web.bind.annotation.PathVariable;
10 import org.springframework.web.bind.annotation.RequestBody;
11 import org.springframework.web.bind.annotation.RequestMapping;
12 import org.springframework.web.bind.annotation.RequestMethod;
13 import org.springframework.web.bind.annotation.ResponseBody;
14 import org.springframework.web.bind.annotation.RestController;
15
16 @RestController
17 @RequestMapping("/teams/{teamId}/players")
18 public class PlayerController {
19
20     @Autowired
21     private PlayerRepository playerRepository;
22
23     public PlayerController() {
24 }
```

```

25
26     public PlayerController(PlayerRepository playerRepository) {
27         this.playerRepository = playerRepository;
28     }
29
30     @RequestMapping(method = RequestMethod.GET)
31     public @ResponseBody
32     List<Player> getTeamPlayers(@PathVariable int teamId) {
33         return this.playerRepository.getByTeam(teamId);
34     }
35
36     @RequestMapping(value = "{playerId}", method = RequestMethod.GET)
37     public @ResponseBody
38     Player getById(@PathVariable int teamId, @PathVariable int playerId) {
39         return this.playerRepository.get(teamId, playerId);
40     }
41
42     @RequestMapping(method = RequestMethod.POST)
43     public @ResponseBody
44     ResponseEntity<Player> create(@PathVariable int teamId, @RequestBody Player
45                                     player) {
46         player.setTeamId(teamId);
47         this.playerRepository.add(player);
48         return new ResponseEntity<>(player, HttpStatus.CREATED);
49     }
50
51     @RequestMapping(method = RequestMethod.PUT)
52     public @ResponseBody
53     ResponseEntity<Player> update(@PathVariable int teamId, @RequestBody Player
54                                     player) {
55         player.setTeamId(teamId);
56         this.playerRepository.update(player);
57         return new ResponseEntity<>(player, HttpStatus.OK);
58     }
59
60     @RequestMapping(value = "{playerId}", method = RequestMethod.DELETE)
61     public @ResponseBody
62     ResponseEntity<Player> delete(@PathVariable int teamId, @PathVariable int
63                                     playerId) {
64         this.playerRepository.delete(teamId, playerId);
65         return new ResponseEntity<>(HttpStatus.OK);
66     }

```

2.4 Què s'ha après?

En aquest apartat heu vist les bases per al desenvolupament dels serveis web RESTful amb Spring i les heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après:

- Les nocions bàsiques dels serveis web RESTful amb Spring.
- Desenvolupar, desplegar i provar un servei web RESTful senzill amb Spring.
- Fer tests unitaris dels serveis web RESTful amb Spring.
- Desenvolupar, desplegar i provar un servei web RESTful complex amb Spring que inclou operacions CRUD sobre un recurs.

Per aprofundir en aquests conceptes i veure quins són els conceptes que hi ha darrere de HATEOAS (de l'anglès *Hypermedia as the Engine of Application State*) us recomanem la realització de l' activitat associada a aquest apartat.

3. Serveis web RESTful amb Spring. Consumint serveis web

Explicarem, mitjançant exemples, com podem consumir serveis web RESTful remots amb diferents tipus de clients.

Es poden provar els serveis RESTful **amb qualsevol eina que permeti fer peticions HTTP**.

La primera eina en la qual pensaríem tots és un navegador web (Microsoft Internet Explorer, Google Chrome, Mozilla Firefox, etc.). El problema és que els navegadors, per defecte, tan sols poden fer peticions GET i peticions POST. Si voleu fer peticions d'un altre tipus (PUT, DELETE, HEAD) caldrà instal·lar algun *plugin* al navegador que us ho permeti (Postman és un possible *plugin* per a Google Chrome, però n'hi ha molts).

Una altra opció és emprar la utilitat **cURL**, que us permet fer tot tipus de peticions HTTP per línia de comandes.

Si el que volem és consumir serveis web RESTful des de Java ho podem fer amb l'API de baix nivell `java.net.HttpURLConnection` o alguna llibreria propietària que us fes la vida una mica més fàcil. Una opció és utilitzar la classe `RestTemplate`, que proporciona Spring, o bé l'API client de JAX-RS; les dues opcions permeten fer tota mena de peticions HTTP als serveis web RESTful remots de forma fàcil.

Spring proporciona la classe `RestTemplate` per **simplificar la comunicació** HTTP entre el client i el servidor. `RestTemplate` permet fer tota mena de peticions HTTP als serveis web RESTful remots de forma fàcil.

`RestTemplate` es pot utilitzar per fer tests d'integració per provar els serveis RESTful, **estiguin o no** desenvolupats amb Spring.

AJAX és un conjunt de tècniques de desenvolupament d'aplicacions web que permeten crear aplicacions web client asíncrones que cada vegada s'utilitzen més per accedir a serveis web RESTful, tant des d'aplicacions web com des d'aplicacions mòbils.

3.1 Un client Java per al servei web RESTful ""Hello, World!!!"

Veurem com consumir serveis web RESTful des d'una aplicació Java *stand-alone* utilitzant la classe `RestTemplate` que proporciona Spring per fer-ho.

3.1.1 Creació i configuració inicial del projecte

El projecte té un servei web RESTful configurat a la classe `GreetingController` amb el següent codi:

```

1 package cat.xtec.ioc.springresthelloioc.controller;
2
3 import cat.xtec.ioc.springresthelloioc.domain.Greeting;
4 import java.util.concurrent.atomic.AtomicLong;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.bind.annotation.RequestParam;
8 import org.springframework.web.bind.annotation.RestController;
9
10 @RestController
11 public class GreetingController {
12
13     private static final String template = "Hello, %s";
14     private final AtomicLong counter = new AtomicLong();
15
16     @RequestMapping(method = RequestMethod.GET, value = "/hello")
17     public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!"
18                             ) String name) {
19         return new Greeting(counter.incrementAndGet(),
20                             String.format(template, name));
21     }
22 }
```

Aquest és un servei web RESTful desenvolupat amb Spring que respon a peticions GET a l'URI `/hello` amb `"Hello, World!!!"` si no li passem cap paràmetre, i si li passem un paràmetre anomenat `name` torna una salutació personalitzada canviant la paraula `World` pel nom especificat a `name`.

El servei web torna la salutació en format JSON i respon a les peticions GET a `/hello`:

```
1 {"id":1,"content":" Hello, World!!!"}
```

I a les peticions GET a `/hello?name=User` amb:

```
1 {"id":1,"content":"Hello, User!"}
```

Un cop importat el projecte cal que modifiqueu el `pom.xml` per afegir les dependències cap a Jackson per tal que Spring pugui fer la transformació de dades de JSON a objecte Java. Per fer-ho, afegiu les següents línies al fitxer `pom.xml`:

```

1 <dependency>
2     <groupId>com.fasterxml.jackson.core</groupId>
3     <artifactId>jackson-databind</artifactId>
```

Descarregueu el codi del projecte "Springresthellojavaclientioc" en l'estat inicial d'aquest apartat en l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

```

4 <version>2.5.3</version>
5 </dependency>
```

Si recarregueu el pom.xml fent clic amb el botó dret damunt el nom del projecte i prement l'opció *Reload POM* del menú contextual ja tindreu el projecte configurat i llest per començar a crear el client Spring.

3.1.2 Creació del client Java 'stand-alone'

El primer que necessitem és una classe Java que representarà el model del domini a la part client i que Spring construirà a partir del missatge JSON retornat pel servei web.

Fixeu-vos que en el nostre cas es podria haver utilitzat per fer això la classe Greeting que teniu al paquet cat.xtec.ioc.springresthelloioc.domain, i ens estalviaríem de crear-ne una de nova. En crearem una, ja que l'habitual és que el client i el servei web no estiguin al mateix projecte; nosaltres els hem posat al mateix projecte per simplicitat.

Creeu, doncs, la classe Java que modelarà la resposta del servei web al paquet cat.xtec.ioc.springresthelloioc.client i l'anomeneu GreetingClient:

```

1 package cat.xtec.ioc.springresthelloioc.client;
2
3 import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
4
5 @JsonIgnoreProperties(ignoreUnknown = true)
6 public class GreetingClient {
7
8     private long id;
9     private String content;
10
11    public long getId() {
12        return id;
13    }
14
15    public void setId(long id) {
16        this.id = id;
17    }
18
19    public String getContent() {
20        return content;
21    }
22
23    public void setContent(String content) {
24        this.content = content;
25    }
26
27    @Override
28    public String toString() {
29        return "Greeting{" +
30            "id='" + id + '\'' +
31            ", content='" + content +
32            '}';
33    }
34}
```

Es tracta d'una classe Java normal amb una propietat per a cada un dels camps que conté la salutació que torna el servei web. L'única particularitat és l'anotació `@JsonIgnoreProperties(ignoreUnknown = true)` per indicar que en la transformació de JSON a objecte Java ignori les propietats que no tinguin un mapatge definit.

Creeu la classe Java que farà de client al paquet `cat.xtec.ioc.springresthelloioc.client` i l'anomeneu `HelloWorldClient`:

```

1 package cat.xtec.ioc.springresthelloioc.client;
2
3 import org.springframework.web.client.RestTemplate;
4
5 public class HelloWorldClient {
6
7     public static void main(String[] args) {
8         RestTemplate restTemplate = new RestTemplate();
9         GreetingClient greeting = restTemplate.getForObject("http://localhost
10             :8080/springresthellojavaclientioc/hello", GreetingClient.class);
11         System.out.println(greeting.toString());
12         greeting = restTemplate.getForObject("http://localhost:8080/
13             springresthellojavaclientioc/hello?name=User", GreetingClient.
14             class);
15         System.out.println(greeting.toString());
16     }
17 }
```

Com veieu, el client és molt simple: creem un objecte de tipus `RestTemplate` i cridem el mètode `getForObject` amb l'URL del servei web i la classe en la qual ha de convertir el missatge JSON rebut.

Aquesta és només una de les maneres d'utilitzar la classe `RestTemplate`; n'hi ha moltes més que us permetran fer crides amb tot el conjunt de verbs HTTP, passar paràmetres JSON, recuperar tota la informació de la resposta rebuda, etc. Per veure totes les opcions que us permet la classe `RestTemplate` podeu consultar la seva API en la següent adreça: bit.ly/2nywYZy.

Ara ja només ens queda desplegar el servei web i executar el client per veure si es comporta com volem.

3.1.3 Desplegament del servei web i prova amb el client Java

Desplegueu el servei web com a part de l'aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans.

Comproveu que el servei web està desplegat correctament accedint a l'URL `localhost:8080/springresthellojavaclientioc/hello` amb un navegador. El servei web us ha de tornar la salutació “Hello, World!!!” en format JSON.

Si tot és correcte ja podem executar el client; per fer-ho, poseu-vos damunt de la classe `HelloWorldClient` i feu *Run File* a NetBeans i veureu la salutació ”Hello, World!!!” i ”Hello, User” a la consola de sortida:

```

1 Greeting{id='1', content=Hello, World!!!}
2 Greeting{id='2', content=Hello, User}
3
4 BUILD SUCCESS
5

```

3.2 Tests d'integració per al servei web RESTful '"Hello, World!!!"'

Vegem com fer tests d'integració d'un servei web RESTful amb JUnit i la classe RestTemplate.

El primer que farem serà desplegar a Glassfish el servei web REST que volem provar i després crearem el conjunt de tests d'integració que faran peticions HTTP al servei web.

Els **tests d'integració** difereixen dels tests unitaris, ja que no testegen el codi de forma aïllada sinó que requereixen que el codi a testejar estigui desplegat al servidor d'aplicacions per funcionar.

3.2.1 Creació i configuració inicial del projecte

El projecte té un servei web RESTful configurat a la classe GreetingController amb el següent codi:

```

1 package cat.xtec.ioc.springresthelloioc.controller;
2
3 import cat.xtec.ioc.springresthelloioc.domain.Greeting;
4 import java.util.concurrent.atomic.AtomicLong;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.bind.annotation.RequestParam;
8 import org.springframework.web.bind.annotation.RestController;
9
10 @RestController
11 public class GreetingController {
12
13     private static final String template = "Hello, %s";
14     private final AtomicLong counter = new AtomicLong();
15
16     @RequestMapping(method = RequestMethod.GET, value = "/hello")
17     public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!"
18                             ") String name) {
19         return new Greeting(counter.incrementAndGet(),
20                             String.format(template, name));
21     }
21

```

Descarregueu el codi del projecte "Springtestintresthelloioc" de l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

Aquest és un servei web RESTful desenvolupat amb Spring que respon a peticions GET a l'URI /hello amb "Hello, World!!!" si no li passem cap paràmetre, i si li

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

JUnit

JUnit és un *framework* de test que utilitza anotacions per identificar els mètodes que especificuen un test. A JUnit, un test, ja sigui unitari o d'integració, és un mètode que s'especifica en una classe que només s'utilitza per al test. Això s'anomena una classe de test. Un mètode de test amb JUnit 4 es defineix amb l'anotació `@org.junit.Test`. En aquest mètode s'utilitza un mètode d'asserció en el qual es comprova el resultat esperat de l'execució de codi en comparació del resultat real.

passem un paràmetre anomenat `name` torna una salutació personalitzada canviant la paraula *World* pel nom especificat a `name`.

El servei web torna la salutació en format JSON i respon a les peticions GET a `/hello` amb:

```
1 {"id":1,"content":" Hello, World!!!"}
```

I a les peticions GET a `/hello?name=User` amb:

```
1 {"id":1,"content":"Hello, User"}
```

Crearem els tests d'integració dins el mateix projecte que conté el codi del servei web que volem provar. Una altra opció, igualment vàlida, seria crear un nou projecte i posar-hi només els tests.

Un cop importat el projecte cal que modifiqueu el `pom.xml` per afegir les dependències cap a Jackson per tal que Spring pugui fer la transformació de dades de JSON a objecte Java i a JUnit 4. Per fer-ho, afegiu les següent línies al fitxer `pom.xml`:

```
1 <dependency>
2   <groupId>com.fasterxml.jackson.core</groupId>
3   <artifactId>jackson-databind</artifactId>
4   <version>2.5.3</version>
5 </dependency>
6 <dependency>
7   <groupId>junit</groupId>
8   <artifactId>junit</artifactId>
9   <version>4.12</version>
10  <scope>test</scope>
11  <type>jar</type>
12 </dependency>
```

Si recarregueu el `pom.xml` fent clic amb el botó dret damunt el nom del projecte i premeu l'opció *Reload POM* del menú contextual ja tindreu el projecte configurat i llest per començar a crear el client amb Spring.

Primer desplegueu el servei web com a part de l'aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans, i comproveu que s'ha desplegat correctament accedint a l'URL localhost:8080/springtestintresthelloworld/hello amb un navegador. El servei web us ha de tornar la salutació “*Hello, World!!!*” en format JSON:

```
1 {"id":1,"content":" Hello, World!!!"}
```

Noteu que l'identificador retornat anirà canviant en funció del número de petició.

3.2.2 Creació i execució dels tests d'integració

Ara toca començar a crear els tests d'integració per provar el servei web RESTful.

El primer que necessitem és una classe Java que representarà el model del domini i que Spring construirà a partir del missatge JSON retornat pel servei web.

Fixeu-vos que per fer això es podria haver utilitzat la classe Greeting que teniu al paquet cat.xtec.ioc.springresthelloioc.domain i ens estalviaríem de crear-ne una de nova (tan sols li hauríeu d'haver afegit un constructor sense paràmetres).

Creeu un nou paquet dins de *Test Packages* anomenat, per exemple, cat.xtec.ioc.springresthelloioc.controller, i creeu-hi una classe Java anomenada GreetingTest:

```

1 package cat.xtec.ioc.springresthelloioc.controller;
2
3 import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
4
5 @JsonIgnoreProperties(ignoreUnknown = true)
6 public class GreetingTest {
7
8     private long id;
9     private String content;
10
11    public long getId() {
12        return id;
13    }
14
15    public void setId(long id) {
16        this.id = id;
17    }
18
19    public String getContent() {
20        return content;
21    }
22
23    public void setContent(String content) {
24        this.content = content;
25    }
26
27    @Override
28    public String toString() {
29        return "Greeting{" +
30            "id='" + id + '\'' +
31            ", content=" + content +
32            '}';
33    }
34 }
```

Es tracta d'una classe Java normal amb una propietat per a cada un dels camps que conté la salutació que torna el servei web. L'única particularitat és l'anotació `@JsonIgnoreProperties(ignoreUnknown = true)` per indicar que en la transformació de JSON a objecte Java ignori les propietats que no tinguin un mapatge definit.

Creeu ara al paquet cat.xtec.ioc.springresthelloioc.controller una classe Java anomenada GreetingControllerTest:

```

1 package cat.xtec.ioc.springresthelloioc.controller;
2
3 import javax.ws.rs.core.MediaType;
4 import static org.junit.Assert.assertEquals;
5 import org.junit.Test;
6 import org.springframework.http.HttpStatus;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.web.client.RestTemplate;
```

```

9
10
11 public class GreetingControllerTest {
12     private static final String uri = "http://localhost:8080/
13         springtestintresthelloioc/hello";
14
15 }
```

Estructura dels tests

Un dels patrons més utilitzats a l' hora d'estructurar el codi d'un test és l'anomenat AAA (de l' anglès *Arrange-Act-Assert*), amb el qual els tests sempre tindran una fase de **preparació** (*Arrange*), una d'**execució** (*Act*) i una de **verificació** de resultats (*Assert*).

El primer test que farem serà un test que **comprovi que la petició /hello sense cap paràmetre torna la representació JSON de la salutació “Hello, World!!!”**; per fer-ho, creeu un mètode anomenat `greetingShouldReturnHelloWorldWithoutName` dins la classe `GreetingControllerTest` amb el següent codi:

```

1 @Test
2 public void greetingShouldReturnHelloWorldWithoutName() {
3     // Arrange
4     RestTemplate restTemplate = new RestTemplate();
5
6     // Act
7     GreetingTest greeting = restTemplate.getForObject(uri, GreetingTest.class);
8
9     // Assert
10    assertEquals("Hello, World!!!", greeting.getContent());
11 }
```

Hem anotat el mètode `greetingShouldReturnHelloWorldWithoutName` amb l'anotació `@Test` per indicar que es tracta d'un mètode de test; en la fase de **preparació** simplement creem l'objecte `RestTemplate` per enviar les peticions HTTP al servei web RESTful, després **executem** la petició cridant el mètode `getForObject` i, finalment, **verifiquem** que la salutació retornada coincideix amb la que esperem.

Ja podem executar el test que hem creat fent *Test File* al menú contextual de la classe de *Test*, i si tot ha anat bé veureu el resultat de l'execució correcta dels tests a la finestra de *Test*.

Per mostrar alguna altra capacitat de la classe `RestTemplate` farem un altre test que comprovarà que el codi HTTP de retorn és un 200 (OK) i que el *content type* sigui JSON.

Per fer-ho, creeu un mètode anomenat `greetingShouldReturnOKAndJSON` dins la classe `GreetingControllerTest` amb el següent codi:

```

1 @Test
2 public void greetingShouldReturnOKAndJSON () {
3     // Arrange
4     RestTemplate restTemplate = new RestTemplate();
5
6     // Act
7     ResponseEntity<GreetingTest> response = restTemplate.getEntity(uri,
8         GreetingTest.class);
9
10    // Assert
11    assertEquals(HttpStatus.OK, response.getStatusCode());
12    assertEquals(MediaType.APPLICATION_JSON, response.getHeaders().getContentType
13        ().get("Content-Type"));
14    assertEquals(MediaType.APPLICATION_JSON, response.getHeaders().getContentType
15        ());
16 }
```

Executeu-lo de la mateixa manera i comproveu que el test s'executa correctament.

El darrer test que farem serà un test que **comprovi que la petició `/hello?name=User` torna la representació JSON de la salutació “Hello, User”**; per fer-ho, creeu un mètode anomenat `greetingShouldReturnHelloNameWithName` dins la classe `GreetingControllerTest` amb el següent codi:

```
1 @Test
2 public void greetingShouldReturnHelloNameWithName() {
3     // Arrange
4     RestTemplate restTemplate = new RestTemplate();
5
6     // Act
7     GreetingTest greeting = restTemplate.getForObject(uri + "?name=User",
8             GreetingTest.class);
9
10    // Assert
11    assertEquals("Hello, User", greeting.getContent());
12 }
```

El codi és molt similar al primer test; executeu-lo de la mateixa manera i comproveu que el test s'executa correctament.

Aquests tres exemples us donen la base per tal que pugueu fer tots els tests d'integració que se us acudeixin i així tenir els serveis web RESTful totalment provats!

L'ús de `RestTemplate` és l'opció que proporciona Spring; els mateixos tests d'integració els podríeu haver escrit amb qualsevol altre bastiment que permeti fer peticions HTTP. Una opció perfectament vàlida hauria estat utilitzar **JAX-RS**, l'API client per consumir serveis web RESTful que proporciona Java EE.

Noteu que, a part dels tests d'integració, Spring també us permet fer tests unitaris per provar els serveis web RESTful.

3.3 Un client JavaScript per al servei web RESTful ”Hello, World!!!”

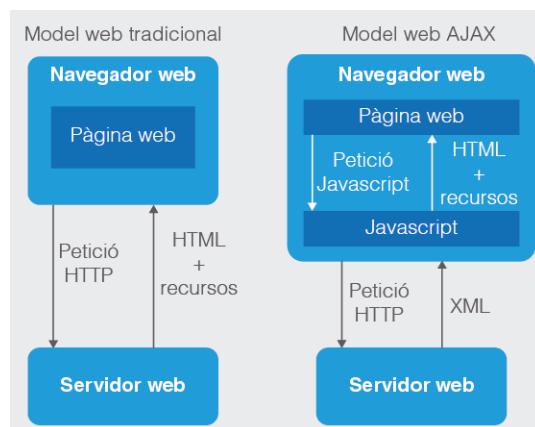
Veurem com consumir serveis web RESTful des d'una aplicació web amb peticions AJAX de JavaScript. Utilitzarem Angular JS com a bastiment JavaScript per fer les peticions AJAX.

AJAX (de l'anglès *Asynchronous JavaScript And XML*) és una tècnica de desenvolupament que permet fer les pàgines web d'una aplicació web interactives amb JavaScript.

AJAX és un conjunt de tècniques (vegeu la figura 3.1) de desenvolupament d'aplicacions web que permeten crear aplicacions web client asíncrones. Amb

AJAX, la part client de les aplicacions web pot enviar i recuperar informació del servidor de forma asíncrona (en *background*) sense interferir en com es mostra i com es comporta la pàgina. Aquesta forma de desacoblar la capa d'intercanvi de dades de la capa de presentació permet a les pàgines web, i, per extensió, a les aplicacions web, intercanviar contingut dinàmicament amb el servidor sense haver de recarregar tota la pàgina. Tot i que inicialment estava centrat en XML, cada vegada més el format de les dades intercanviades tendeix més a ser JSON, ja que aquest és un format nadiu per a les dades en JavaScript.

FIGURA 3.1. AJAX



Objectiu de l'exemple

L'objectiu d'aquest exemple no és aprofundir en el coneixement d'Angular JS, sinó que només pretén mostrar com es pot fer servir AJAX com a client per consumir serveis web RESTful.

Si voleu aprofundir en el coneixement d'Angular JS ho podeu fer a l'URL angularjs.org.

Les aplicacions que fan servir AJAX segueixen, en molta mesura, els principis de disseny inherents a l'arquitectura d'aplicacions REST. Podem veure cada una de les peticions AJAX com una petició a un servei REST que tornarà les dades en un format determinat, moltes vegades JSON, per tal que l'aplicació web les tracti i mostri a l'usuari sense necessitat de recarregar tota la pàgina.

Aquestes característiques fan d'AJAX una de les formes més utilitzades a l'hora de consumir serveis web RESTful des de pàgines o aplicacions web.

Actualment hi ha multitud de bastiments JavaScript que proporcionen, entre moltes altres coses, la possibilitat de fer peticions AJAX. Potser el més popular des dels seus inicis és jQuery.

Angular JS és un altre bastiment JavaScript que, entre moltes altres coses, proporciona capacitats i ajudes per fer peticions AJAX a serveis web RESTful.

A l'exemple veurem un client Angular JS que consumirà amb AJAX el servei web RESTful de salutacions.

3.3.1 Creació i configuració inicial del projecte

Descarregueu el codi del projecte "Springresthelloangularclientioc" en l'estat inicial d'aquest apartat en l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

El projecte té un servei web RESTful configurat a la classe GreetingController amb el següent codi:

```
1 package cat.xtec.ioc.springresthelloioc.controller;
```

```
2
```

```

3 import cat.xtec.ioc.springresthelloioc.domain.Greeting;
4 import java.util.concurrent.atomic.AtomicLong;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.bind.annotation.RequestParam;
8 import org.springframework.web.bind.annotation.RestController;
9
10 @RestController
11 public class GreetingController {
12
13     private static final String template = "Hello, %s";
14     private final AtomicLong counter = new AtomicLong();
15
16     @RequestMapping(method = RequestMethod.GET, value = "/hello")
17     public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!"
18         ") String name) {
19         return new Greeting(counter.incrementAndGet(),
20             String.format(template, name));
21     }
}

```

Aquest servei web és un servei web RESTful desenvolupat amb Spring que respon a peticions GET a l'URI */hello* amb “Hello, World!!!” si no li passem cap paràmetre, i si li passem un paràmetre anomenat `name` torna una salutació personalitzada canviant la paraula *World* pel nom especificat a `name`.

El servei web torna la salutació en format JSON i respon a les peticions GET a */hello* amb:

```
1 {"id":1,"content":" Hello, World!!!"}
```

I a les peticions GET a */hello?name=User* amb:

```
1 {"id":1,"content":"Hello, User!"}
```

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

3.3.2 Creació del client JavaScrip amb Angular JS

El primer que necessitem és crear un mòdul controlador Angular JS que consumeixi el servei web. Per fer-ho, creeu un fitxer anomenat `helloangular.js` a *Web Pages/static* amb el següent contingut:

```

1 angular.module('demo', [])
2   .controller('Hello', function ($scope, $http) {
3     $http.get('http://localhost:8080/springresthelloangularclientioc/hello').
4       then(function (response) {
5         $scope.greeting = response.data;
6       });
7   });

```

Aquest mòdul controlador, anomenat *Hello*, es representa com una funció JavaScript a la qual se li passa com a paràmetres els components `scope` i `http`. La funció fa servir el component `http` per fer una crida al servei RESTful a */hello*.

Si la crida té èxit, s'assignarà el JSON retornat des del servei web la variable `scope.greeting` amb la creació d'un objecte JavaScript que representarà el

model. L'establiment d'aquest objecte al model fa que Angular el pugui utilitzar al DOM de la pàgina que fa la sol·licitud i mostrar el seu contingut a l'usuari.

Un cop tenim el controlador ens cal crear la pàgina HTML que el carregarà al navegador de l'usuari. Per fer-ho, creeu un fitxer anomenat helloangular.html a *Web Pages/static* amb el següent contingut:

```

1  <!doctype html>
2  <html ng-app="demo">
3      <head>
4          <title>Hello AngularJS</title>
5          <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/
6              css/bootstrap.min.css">
7          <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.3/
8              angular.min.js"></script>
9          <script src="helloangular.js"></script>
10     </head>
11
12     <body>
13         <div ng-controller="Hello">
14             <p>The ID is {{greeting.id}}</p>
15             <p>The content is {{greeting.content}}</p>
16         </div>
17     </body>
18 </html>
```

A la secció *head* de la pàgina hi posem aquestes dues etiquetes:

```

1  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.3/angular.min.
2      js"></script>
3  <script src="helloangular.js"></script>
```

La primera descarrega la llibreria angular del CDN (de l'anglès *Content Delivery Network*) per tal que no l'haguem d'incorporar al projecte, i el segon carrega el codi JavaScript del controlador que hem creat (helloangular.js).

Angular JS habilita un conjunt d'etiquetes pròpies que podrem utilitzar a les pàgines HTML. A l'exemple fem servir l'atribut *ng-app* per indicar que la pàgina és una aplicació Angular JS i l'etiqueta *ng-controller* per indicar que el controlador de la pàgina serà el controlador Hello definit al fitxer JavaScript helloangular.js.

Finalment, la pàgina accedeix a l'objecte del model greeting que ens ha deixat el controlador i mostra l'identificador i la salutació.

Ara ja sols ens queda desplegar el servei web i el client per veure si es comporta com volem.

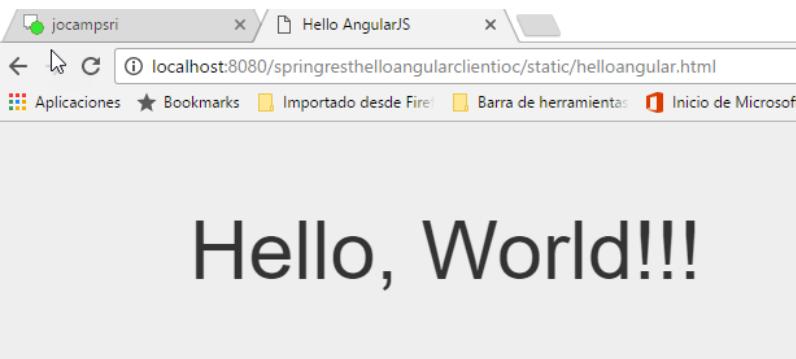
3.3.3 Desplegament del servei web i prova del client Angular

Desplegueu el servei web com a part de l'aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans.

Comproveu que el servei web està desplegat correctament accedint a l'URL localhost:8080/springresthelloangularclientioc/hello amb un navegador. El servei web us ha de tornar la salutació “Hello, World!!!” en format JSON.

Si tot és correcte ja podem provar el client Angular JS accedint a l'URL localhost:8080/springresthelloangularclientioc/static/helloangular.html amb un navegador, i veureu la salutació “Hello, World!!!” tornada pel servei web (vegeu la figura 3.2).

FIGURA 3.2. Execució del client Angular JS



3.4 Què s'ha après?

En aquest apartat heu vist les bases per al desenvolupament de clients Java i JavaScript que accedeixin a serveis web RESTful, i els heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après a:

- Desenvolupar i provar un client Java *stand-alone* que us permeti consultar un servei web RESTful mitjançant la classe RestTemplate.
- Fer tests d'integració amb RestTemplate que us permetin provar els serveis web RESTful.
- Fer crides AJAX a serveis web RESTful.

Per veure com habilitar sol·licituds Cross-Origin per un servei web RESTful us recomanem la realització de l'activitat associada a aquest apartat.