Curso avanzado de JAVA

Manual práctico

Ismael López Quintero

1 - Instalación del entorno de desarrollo
2 - Lenguaje Java y POO
3 - Instalación del entorno servidor de aplicaciones Oracle *Weblogic* Server
4 - EJB
5 - JSP y *servlets*
6 - Implementación de MVC: JSF.
7 - Servicios web
8 - *Websockets*
9 - Ejercicios completos resueltos

Alfaomega

Altaria publicaciones

Revisado por: Sonia Vives y Anna Simón

Datos catalográficos
López, Ismael
Curso avanzado de JAVA JEE. Manual práctico
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México

ISBN: 978-607-622-853-1

Formato: 17 x 23 cm Páginas: 336

Curso avanzado de JAVA JEE. Manual práctico
Ismael López Quintero
ISBN: 978-84-945683-2-9, edición en español publicada por Publicaciones Altaria S.L., Tarragona, España

Primera edición: Alfaomega Grupo Editor, México, febrero 2017

Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, 06720, Ciudad de México.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-622-853-1

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele.

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, Ciudad de México - C.P. 06720, – Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396 – E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia, Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Av. Córdoba 1215 piso 10, C.P. 1055, Buenos Aires, Argentina. – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

A las personas que quiero y que pacientemente
me animan día a día.

Índice general

¿A quién va dirigido el libro?

Este libro está dirigido a:

- Desarrolladores de *software* con un nivel avanzado de programación.

Se da por hecho que el lector del libro debe conocer en profundidad la programación estructurada: los tipos de datos básicos, las estructuras de control, los procedimientos, las funciones; y los tipos de datos complejos, los registros y las listas.

También, es recomendable que el lector tenga conocimientos de programación orientada a objetos, y que sean familiares para él conceptos tales como *clases*, *atributos*, *métodos*, *herencia* e *interfaces*. Si éste no fuera el caso, se recomienda realizar un curso previo de programación orientada a objetos. Aunque se aborda un tema de repaso de programación orientada a objetos, se hace de forma muy somera y como introducción para los temas posteriores del libro.

Asimismo, es recomendable que se tenga experiencia previa con el lenguaje Java, aunque sería suficiente con conocimiento previo del lenguaje C++, debido a la similitud de su sintaxis con Java.

Cómo se ha estructurado el libro

A la hora de redactar un manual sobre Java, se plantea un universo de posibilidades debido a que Java en sí es un universo. Podemos plantear un manual enfocado al lector nobel con conocimientos de programación incipientes o para un lector que quiere desarrollar aplicaciones de escritorio mediante la biblioteca gráfica Swing. Si lo planteamos de alguna de las formas anteriores, estamos dejando de centrarnos en la gran fortaleza del lenguaje Java. Dicha fortaleza se llama *JEE* (*Java Enterprise Edition* o Java Edición Empresarial), que permite desarrollar arquitecturas de *software* distribuidas en las que los sistemas locales se intercomunican con otros sistemas gracias al conjunto de herramientas que provee dicha plataforma.

El manual que tiene ante sí se centra en estudiar de forma generalista las principales características de JEE.

Tras instalar el entorno de desarrollo necesario para poder empezar a desarrollar aplicaciones Java en su edición estándar, estudiamos en un capítulo, que podríamos llamar *de repaso*, las características de la programación orientada a objetos y su implementación en Java, como paso inicial para poder empezar a estudiar la JEE. En el cuarto capítulo vemos la instalación del servidor de aplicaciones Oracle WebLogic Server, solución escogida en este manual por su robustez y su facilidad de uso. Posteriormente, nos centramos en la JEE en sí, con capítulos dedicados a los bloques que componen una aplicación empresarial y cómo se despliegan en el servidor, tanto en el contenedor

de EJB como en el contenedor de *servlets*. Estudiamos los componentes *software* que se pueden desplegar en el contenedor de EJB (llamados del mismo modo, *EJB*), así como el corazón de los componentes del contenedor de *servlets*: las páginas JSP y los *servlets*. En un capítulo posterior estudiamos el *framework* de la capa de presentación que hace más fácil el trabajo con páginas JSP y con *servlets*: JSF. Sin centrarnos en ninguna subimplementación de componentes, estudiamos JSF de forma generalista. Para la conectividad de las aplicaciones dedicamos dos capítulos finales, uno centrado en los servicios web RESTful y su implementación con JAX-RS (librería Jersey); y otro capítulo centrado en los *websockets*.

Los servicios web son una solución que permite interconectar sistemas de datos a través de Internet. La solución REST (transferencia del estado representacional) se sirve del protocolo HTTP para poder crear una API sin estado que sirva de acceso universal al *back-end* de nuestras aplicaciones, pudiendo crear cualquier tipo de clientes, por ejemplo, clientes móviles. Con los *websockets* podemos crear aplicaciones web en las que los diversos usuarios que están conectados a nuestra aplicación pueden mantener sesiones activas y ser informados del estado de la aplicación en tiempo real. En definitiva, un manual sobre JEE que pretende ser generalista y a la vez concreto con las tecnologías explicadas que nos encontramos en el estudio.

Introducción

CAPÍTULO 1

La JEE es el marco de trabajo de las aplicaciones empresariales escritas en lenguaje Java. Muchos desarrolladores que se enfrentan a esta tecnología por primera vez encuentran un conjunto de conceptos similares y, del mismo modo, confusos. Conceptos tales como *lenguaje de programación*, *entorno de ejecución*, *kit de desarrollo* o *plataforma Java* están presentes en los manuales que estudian los noveles en esta materia. Java en sí es el lenguaje formal en el que se escriben las aplicaciones que podrán ejecutarse en cualquier máquina en que esté instalado el entorno de ejecución (también conocido como *máquina virtual*) Java. Objeto de este manual es aclarar todos estos conceptos. Igualmente, el manual encaminará su contenido a conocer lo que es el superconjunto de la plataforma Java, también conocido como *Java Edición Empresarial* (JEE). Aprenderá a escribir aplicaciones web en lenguaje Java, así como a poner a disposición de la comunidad de desarrolladores servicios web que puedan acceder al *back-end* de sus aplicaciones.

Java es un lenguaje de programación que nació en el año 1995 de la mano de su diseñador James Gosling y en el seno de la compañía Sun Microsystems. Lo que se pretendió fue diseñar un lenguaje orientado a objetos, multiplataforma, cuyas aplicaciones fuesen fácilmente accesibles a través de Internet. Del mismo modo, se pretendió aislar al programador del problema de la liberación de memoria (problema presente en los lenguajes previos y que más han influido a la sintaxis de Java: C estándar y C++). Veamos someramente cada uno de estos conceptos de Java:

- **Orientado a objetos.** Durante la década de los ochenta y a comienzos de la década de los noventa, el paradigma de programación imperante era el estructurado. Con dicho paradigma, los programas eran difíciles de mantener a medida

que el número de líneas de código crecía; eran difíciles de mantener en el tiempo cuando los *bugs* (fallos) se hacían presentes, ya que los módulos eran interdependientes y no se aplicaban los patrones de diseño orientados a objetos actuales. El paradigma de los objetos se planteó como una alternativa al paradigma estructurado, ya que permite encapsular en "clases" la funcionalidad aislada en distintos módulos de la aplicación. Los módulos dejan de ser tan dependientes unos de otros y es más fácil localizar los fallos que se encuentren en el *software*. Además, comenzaron a aparecer los conocidos *patrones de diseño orientados a objetos*, los cuales ofrecen soluciones elegantes a problemas comunes en el diseño del *software*. Hasta la salida al mercado de Java, la mayor aproximación a la programación orientada a objetos se produjo con C++.

- **Multiplataforma.** Se conoce como *plataforma* a la tupla compuesta por la arquitectura de la máquina y el sistema operativo que la hace funcionar. En muchos lenguajes de programación tradicionales había que recompilar el código fuente para cada arquitectura en la que se iba a ejecutar el *software* concreto. Dicha traducción a código máquina debía hacerse considerando que existiera el compilador para dicho lenguaje en el sistema operativo concreto. Con Java se pretendió diseñar un lenguaje que sólo necesitase compilarse una vez para una máquina universal, conocida como *máquina virtual Java* (JRE, *Java Runtime Environment* o Entorno de Ejecución Java). Bajo esta nueva filosofía, el programador no sólo escribe una vez la aplicación, sino que la traduce a código máquina una sola vez. El código máquina u objeto que se compila se denomina *bytecode*, encapsulándose en ficheros ".class". Es ahora competencia del fabricante de la máquina real en la que vaya a ejecutarse el *bytecode* proporcionar un traductor correcto desde la máquina virtual Java hacia la máquina real. Por este mismo motivo decimos que Java es a la vez un lenguaje compilado e interpretado. Se compila una vez en código comprensible por el JRE y se interpreta en tiempo de ejecución en la máquina real en la que se ha instalado el JRE.

- **Aplicaciones fácilmente accesibles a través de Internet.** Desde el comienzo de Java se implementó su sintaxis y sus características pensando en la red de redes. Los *applets* de Java fueron una solución en la que, a través del navegador y mediante protocolo HTTP, se podrían enviar al navegador aplicaciones cliente que estuvieran en contacto con el servidor que las ha lanzado, creando clientes ricos (clientes a los que se ha trasladado la competencia de ejecutar parte o toda la funcionalidad de la aplicación). Del mismo modo, se diseñó para que la interconexión fuera una de sus mayores virtudes, ideando la JEE en la que los componentes *software* residiesen en contenedores de aplicaciones fácilmente comunicables gracias al sistema de mensajería, la invocación remota de métodos o RMI, CORBA o los servicios web. Todas ellas soluciones pensando en la ejecución distribuida de aplicaciones, incluso en la ejecución distribuida de los módulos de una misma aplicación.

- **Liberación de memoria.** En los lenguajes anteriores más usados por la comunidad de programadores (C estándar y C++), la liberación de memoria debía ser realizada de forma explícita por el desarrollador. El uso de las instrucciones *malloc* y *realloc* en C estándar y de *new* en C++ debía realizarse para la reserva de memoria. Y las instrucciones *free* y *delete* debían invocarse en los dos lenguajes respectivamente para su liberación, haciendo que el manejo de la memoria dinámica fuera algo más que un tedio. Java quiso liberar de este engorro al programador, haciéndole olvidar por completo el asunto de la liberación de memoria. Cuando un bloque de memoria deja de ser apuntado por una referencia, este bloque queda como basura. Existe un servicio en ejecución por la JRE (entorno de ejecución) que se encarga de estudiar los objetos y la memoria usada por éstos, y de vigilar que todos los objetos en memoria estén apuntados al menos por una referencia. Este servicio es el recolector de basura. Cuando algún objeto deja de ser apuntado por al menos una referencia, el recolector de basura se encarga de forma autónoma de dicha liberación. En Java, la liberación de memoria se realiza de forma implícita por parte del entorno de ejecución.

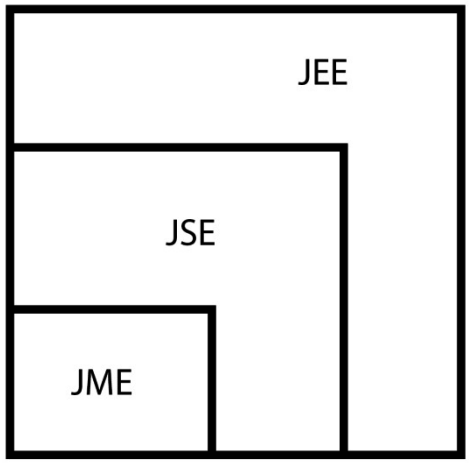
Como se ha comentado, existen un conjunto de conceptos confusos cuando nos enfrentamos al estudio de Java. Algunos de ellos los explicamos brevemente en la siguiente lista:

- **Lenguaje de programación Java.** Es un lenguaje formal, con una sintaxis parecida a la de C++, orientado a objetos, multiplataforma, ideado para aplicaciones en red y con liberación de memoria implícita.
- **Entorno de ejecución o JRE.** Es la máquina virtual en la que se ejecutan las aplicaciones escritas en lenguaje Java. El código Java se compila a un lenguaje intermedio, creando un código máquina conocido como *bytecode*.
- **Kit de desarrollo.** También conocido como *JDK* o *Java Development Kit*, es el conjunto de herramientas que permiten al programador desarrollar aplicaciones en Java. Digamos que es un superconjunto de la JRE. El kit de desarrollo de Java está compuesto por los siguientes componentes (entre otros):
  - "appletviewer.exe": para desarrollo de *applets* (aplicaciones cliente que se lanzan a través de la red).
  - "javac.exe": el compilador y traductor de código Java a *bytecode*.
  - "java.exe": es el punto de entrada al entorno JRE.
  - "javadoc.exe": esta aplicación se usa para desarrollar documentación de los artefactos de Java.
- **Plataforma de Java.** Tradicionalmente se han conocido tres plataformas de Java: la J2ME, la J2SE y la J2EE. Las plataformas de Java son conjuntos de API (*Application Programming Interfaces* o Interfaces de Programación de Aplicaciones), encaminadas al desarrollo de distinto tipo de aplicaciones. Son desarrolladas bajo

el *Java Community Process*, que es un proceso de estandarización de la plataforma Java que permite a las partes interesadas en el desarrollo de las plataformas (empresas, personas físicas o jurídicas) involucrarse en la especificación de las características de dichas plataformas. Las especificaciones se realizan mediante las *Java Specification Request* o JSR. Las tres plataformas mencionadas han cambiado de nombre en las últimas versiones de sus plataformas a JME, JSE y JEE.

- **JME: *Java Micro Edition.*** Es un conjunto de API o librerías encaminadas al desarrollo de aplicaciones que se ejecutarán en máquinas de recursos reducidos, tales como tabletas, teléfonos móviles o PDA. Cabe decir que la JME está obsoleta debido al desarrollo de sistemas operativos nativos para dicho tipo de dispositivos: Android, iOs y Windows Phone, entre otros. Tenemos lenguajes y entornos de desarrollo para programar en dichos sistemas operativos, o soluciones híbridas multiplataforma: Apache Cordova o Adobe PhoneGap.
- **JSE: *Java Standard Edition.*** Colección de librerías necesaria para programar aplicaciones Java en dispositivos de escritorio y servidores. Para desarrollar aplicaciones que se ejecutarán de forma general en una máquina, usaremos la JSE.
- **JEE: *Java Enterprise Edition.*** Es la edición empresarial de Java sobre la que versa este manual que tiene en sus manos. Contiene todas las librerías necesarias para crear aplicaciones distribuidas de N capas, aplicaciones web, servicios web, etc. Las aplicaciones empresariales corren sobre servidores de aplicaciones.

Las tres plataformas mencionadas no son independientes, sino que son complementarias, estando una incluida dentro de otra. El esquema bajo el que se encuadran las tres plataformas es el siguiente:



Podemos ver que la JSE incluye a la JME (aunque obsoleta, sus librerías siguen existiendo). Y que la JEE incluye a la JSE y a la JME.

Este manual que tiene ante usted repasa las nociones básicas del lenguaje Java. Una vez instalado el entorno de desarrollo y repasada la sintaxis básica de Java y su uso, nos centraremos en instalar el servidor de aplicaciones bajo el cual se ejecutarán nuestras aplicaciones: WebLogic Server de Oracle. En el quinto capítulo se mostrará el acceso a datos usando la API de persistencia de Java (JPA). Acto seguido nos centraremos en la lógica de negocio y cómo crear objetos del dominio (*Enterprise JavaBeans* o EJB) que se ejecuten en el servidor de aplicaciones. Dedicamos un capítulo a crear una aplicación web usando el *framework* JSF. Del mismo modo, dedicaremos un capítulo final a crear un servicio web RESTful que sirva para alimentar a la capa de presentación de una aplicación móvil desarrollada bajo la solución híbrida Apache Cordova.

Los prerrequisitos para enfrentarse a la lectura de este manual es un dominio en profundidad de la programación estructurada. También es conveniente que el lector tenga conocimientos de la programación orientada a objetos. En el capítulo 3 se repasan los conceptos fundamentales de este segundo paradigma de programación. También es conveniente que el lector conozca el lenguaje de modelado unificado (UML). El libro no pretende ser un manual de lenguaje Java, sino un manual de uso de Java Edición Empresarial. En el capítulo 3, junto con la programación orientada a objetos, repasaremos sintaxis básica del lenguaje Java que el lector debería dominar.

CAPÍTULO 2

Instalación del entorno de desarrollo

2.1 Instalación

Se mostrará cómo instalar el entorno de desarrollo de Java en un sistema Windows de 64 bits. Para comenzar a desarrollar aplicaciones Java necesitamos dos componentes:

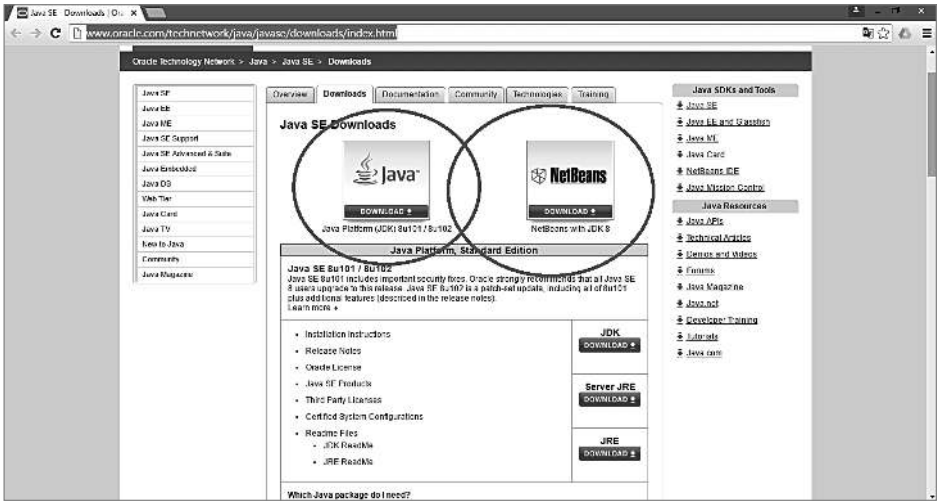
- **JDK *(Java Development Kit)* para la JSE incluyendo la máquina virtual Java (JRE).** Necesitamos dicho componente (JDK) para poder compilar nuestras aplicaciones a *bytecode*, y la JRE para poder ejecutar dicho *bytecode*.
- **Entorno de desarrollo integrado.** Es la aplicación nativa de nuestro sistema operativo que va a permitir tareas cruciales en nuestro desarrollo. Las tareas son:
  - Creación y estructuración de proyectos.
  - Escritura del código fuente.
  - Ejecución del depurador.
  - Compilación.
  - Ejecución de pruebas unitarias.
  - Empaquetado.

Y tantas tareas relacionadas con el desarrollo como se nos ocurran.

Podemos elegir el entorno de desarrollo integrado o IDE *(Integrated Development Environment)* para Java que queramos: Eclipse, NetBeans, BlueJ, JBuilder, JDeveloper... En nuestro caso usaremos NetBeans por dos motivos:

- Es fácil de usar y muy intuitivo.
- Es suministrado por Oracle, la compañía que adquirió Sun Microsystems y que se encarga de empaquetar las plataformas de Java.

Para proceder a la descarga de ambos componentes nos vamos a la siguiente URL: "http://www.oracle.com/technetwork/java/javase/downloads/index.html".



Nos descargamos los ejecutables de dichos componentes.

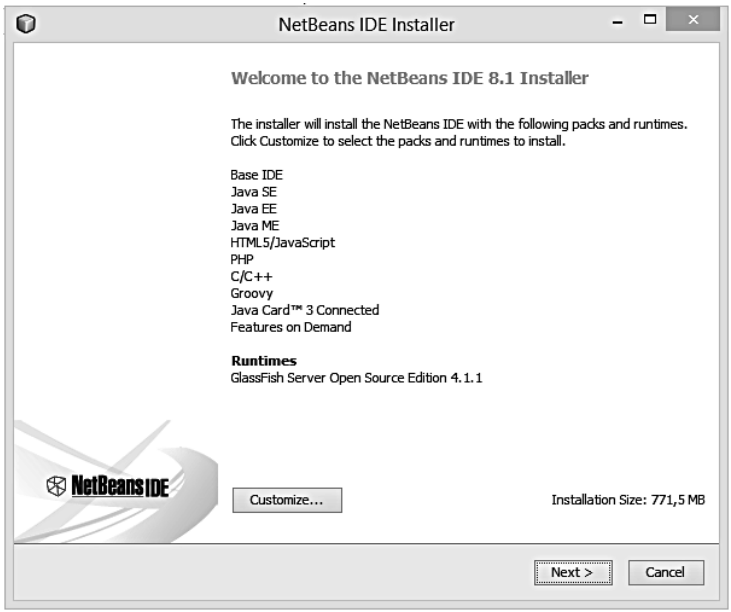
jdk-8u77-windows-x64	05/04/2016 20:32	Aplicación	191.806 KB
netbeans-8.1-windows	05/04/2016 19:41	Aplicación	221.349 KB

Y comenzamos la instalación de la JDK.

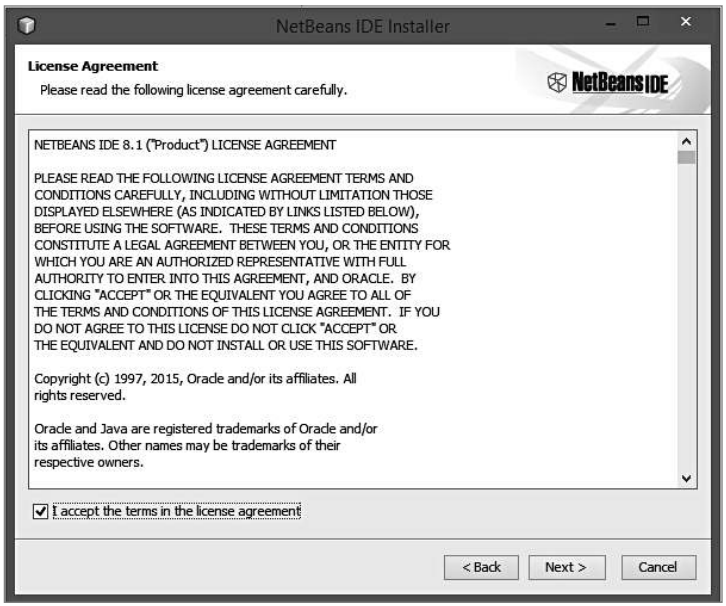
Como hemos indicado, la instalación de la JDK nos solicita la ubicación de la JRE que lleva incorporada. Le decimos que sí a la ubicación por defecto.

Y finalizamos la instalación de la JDK.

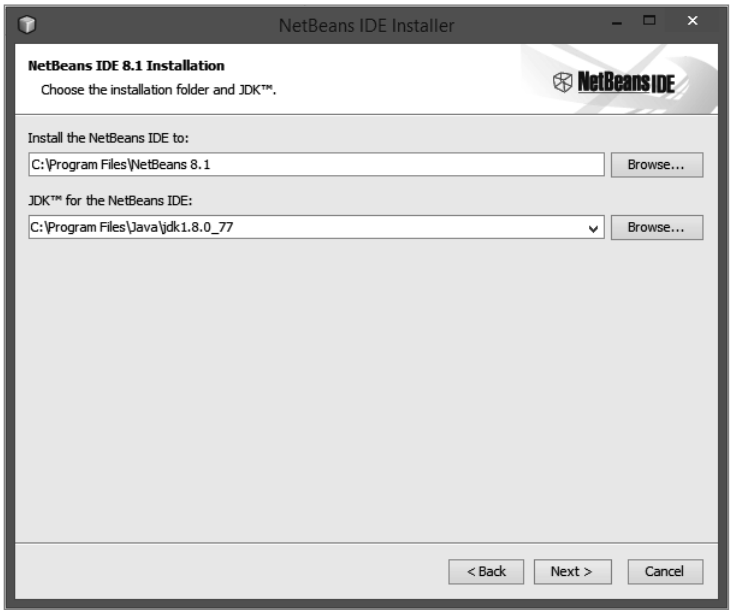
Acto seguido procedemos a la instalación de NetBeans. Podemos observar que la instalación de NetBeans proporciona las plataformas JME, JSE y JEE. Del mismo modo, nos instalará un servidor de aplicaciones de libre distribución para despliegue de aplicaciones JEE: GlassFish.



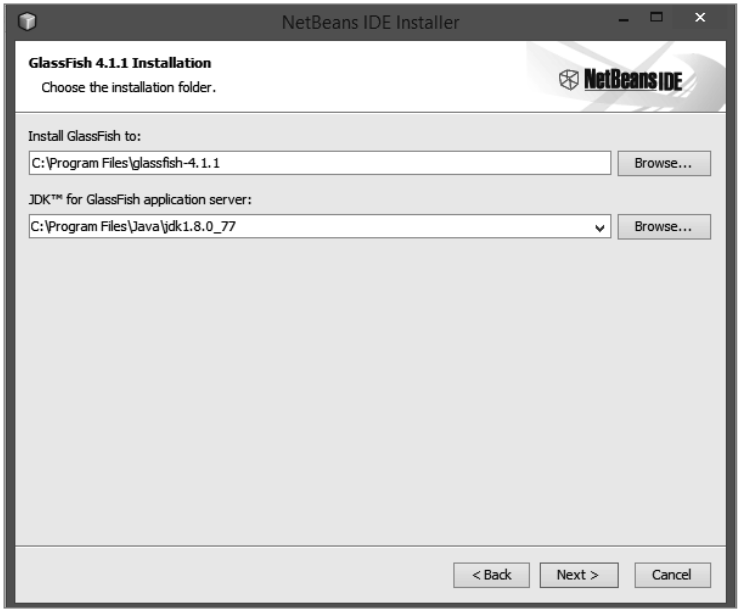
Aceptamos los términos y condiciones.



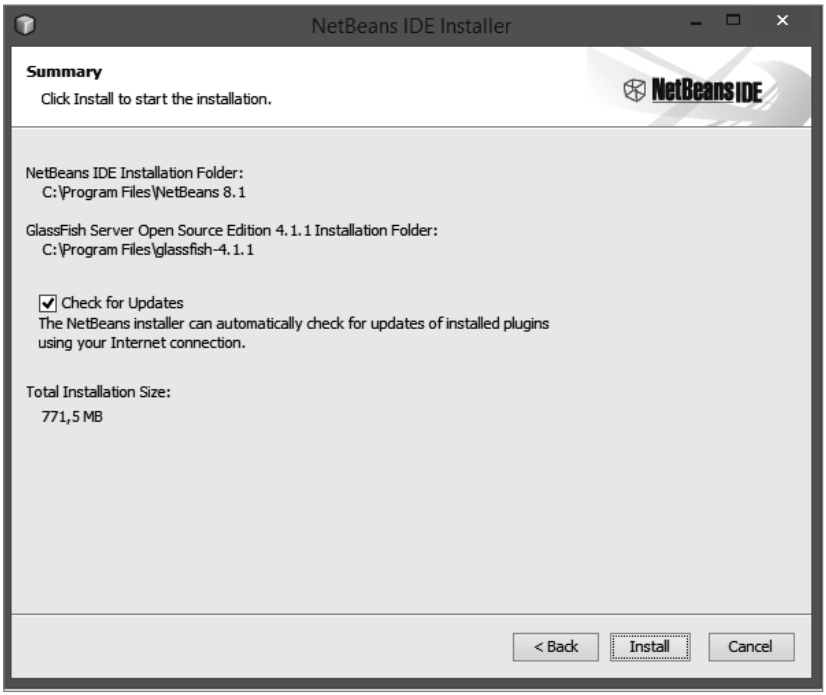
Le indicamos dónde se encuentra la JDK para la JSE instalada previamente.



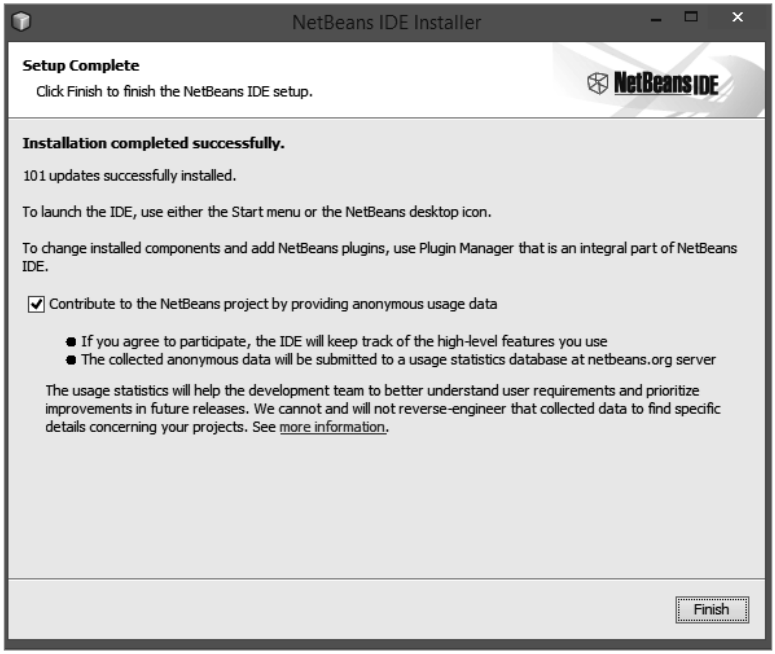
Indicamos la ruta para la instalación del servidor de aplicaciones GlassFish.



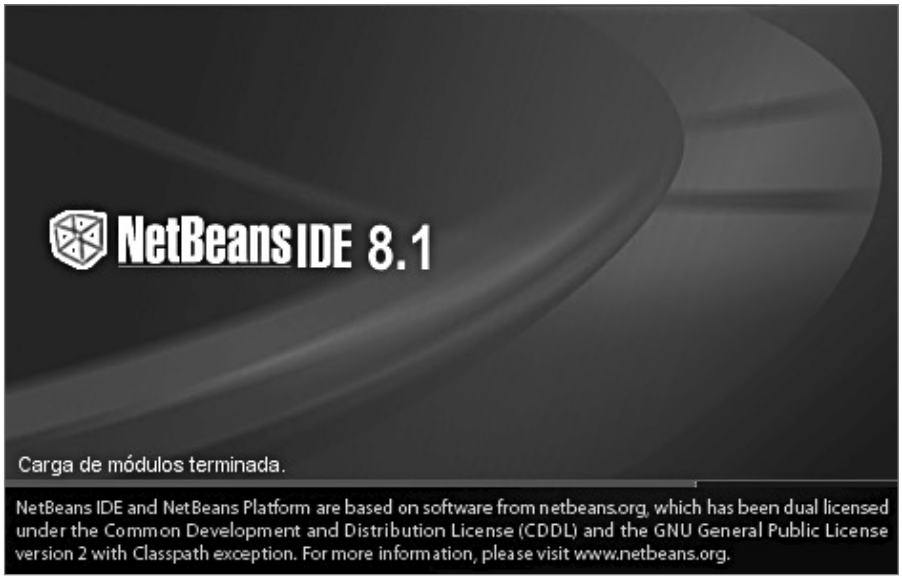
Procedemos a confirmar la instalación del IDE.



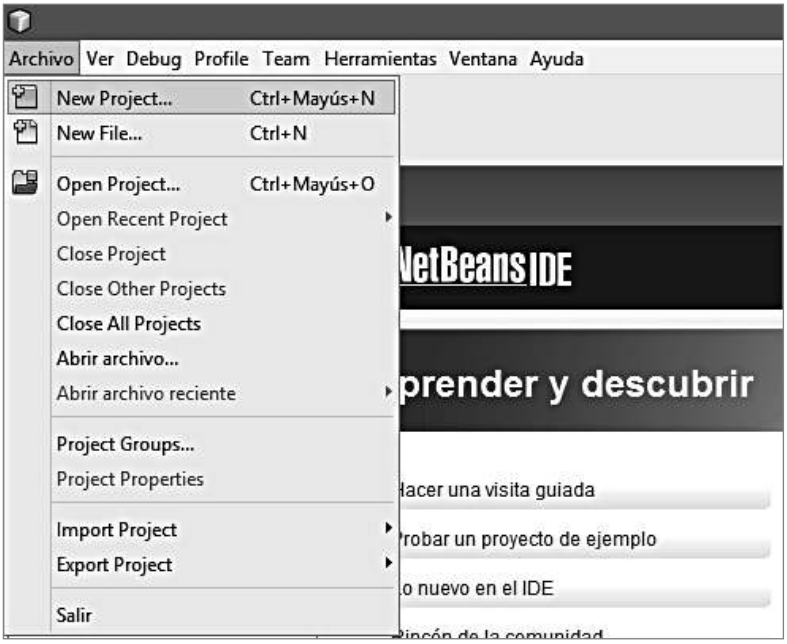
Completamos la instalación.



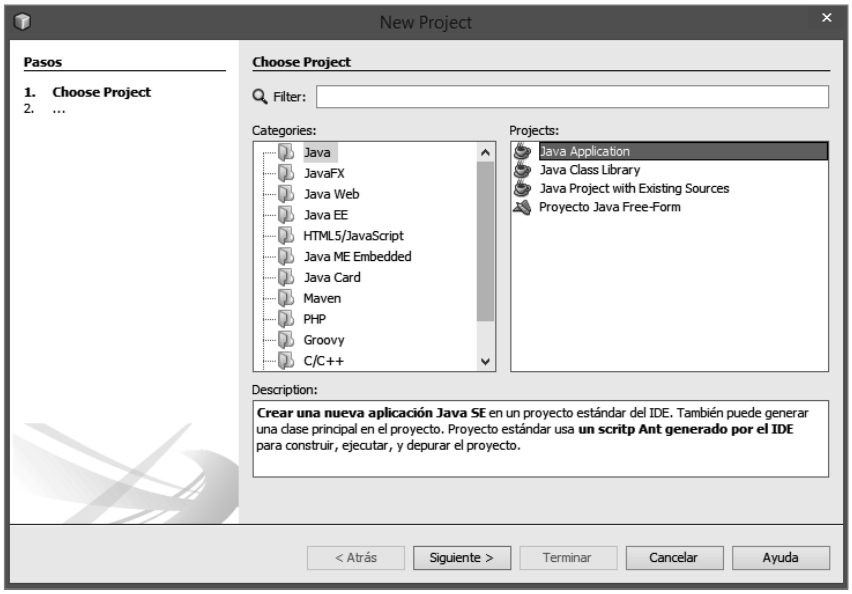
Una vez instalado el entorno de desarrollo, podemos echar a andar NetBeans para desarrollar nuestra primera aplicación Java.



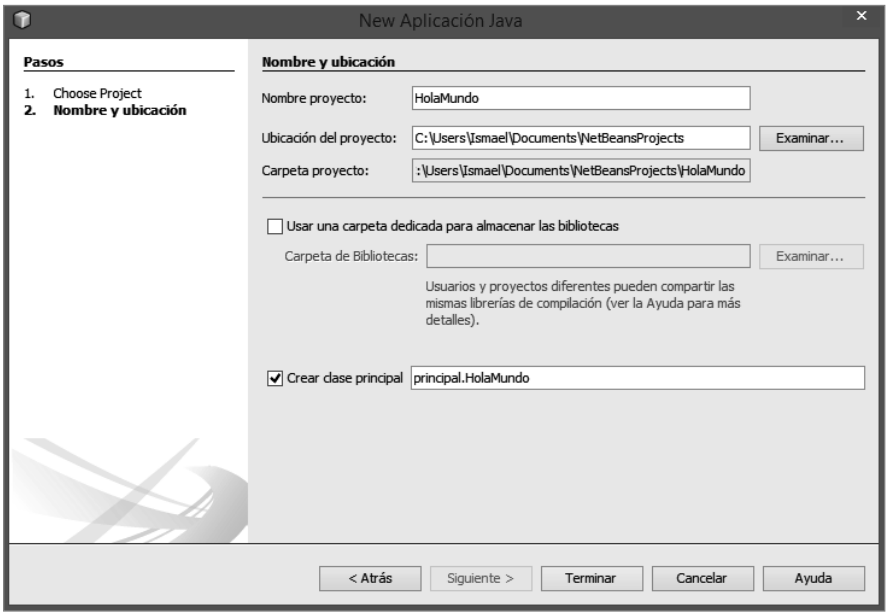
Para crear nuestra primera aplicación Java, seleccionamos la opción "Archivo -> Nuevo Proyecto" *(New Project)*.



Nuestro nuevo proyecto es de tipo Java: "Java Application".



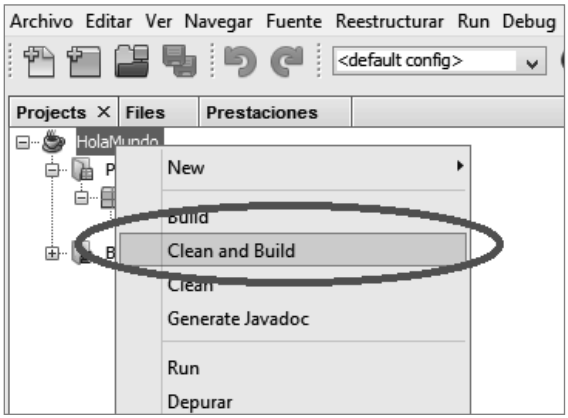
A continuación creamos un nuevo proyecto llamado *HolaMundo*. Va a tener una clase principal con el método *main* en el paquete principal.



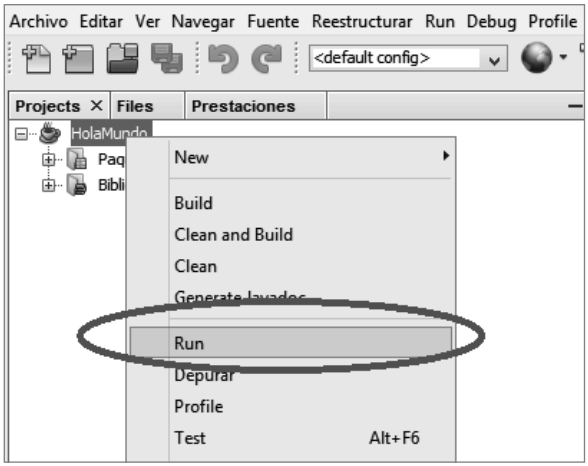
Escribimos el siguiente código en nuestra clase "HolaMundo.java":

```
package principal;
public class HolaMundo {
   public static void main(String[] args) {
      System.out.println("Hola Mundo");
   }
}
```

Con el botón derecho del ratón nos posicionamos sobre el proyecto y lo compilamos: opción "Clean & Build".



Finalmente, lanzamos la aplicación sobre la JRE. Seleccionamos "Run".



La salida por consola de nuestra aplicación será la siguiente:

Hola Mundo

CAPÍTULO 3

¿Qué hay que saber del lenguaje Java y de la POO?

El estudiante que quiera conocer el lenguaje Java debería conocer previamente y en profundidad el paradigma de la programación estructurada. Conceptos tales como *tipos de datos simples y complejos* (registros, listas…), *estructuras de control*, *funciones*, *procedimientos*, *paso de parámetros* y *librerías* deben estar en el argot técnico del lector. Si éste no fuera el caso, se recomienda la lectura de un manual previo de introducción a la programación. Del mismo modo se recomienda que previamente se desarrollen aplicaciones a nivel de programación estructurada. La programación se aprende con muchas horas de máquina.

También es importante que el lector conozca el paradigma de los objetos. Dicho paradigma se conoce como *POO* o *Programación Orientada a Objetos*. De todas formas, en este capítulo repasaremos los siguientes conceptos de programación orientada a objetos, a la vez que veremos su implementación en lenguaje Java, lenguaje orientado a objetos por excelencia. Los conceptos serán los siguientes:

- Clases y objetos.
- Atributos y métodos.
- Método *main*. Archivo Manifest.
- Herencia.
- Polimorfismo.
- Clases abstractas.
- Clases estáticas.
- Implementación de interfaces.
- Excepciones.

3.1 Clases y objetos

Como se comentó en la introducción, las aplicaciones complejas eran difíciles de mantener usando el paradigma estructurado. Para evitar la interconexión de artefactos de código fuente en las aplicaciones con varios miles de líneas de código, se pensó en un nuevo paradigma de programación en el que las unidades de ejecución se asemejaran a los objetos del mundo real. Por ejemplo, si pensamos en una persona (objeto del mundo real), podríamos crear un molde para las personas (a dicho molde lo denominamos *clase*), creando instancias particulares de la persona, denominadas *objetos*. De esta forma, cada objeto persona es responsable de sus propias características y de realizar las acciones correspondientes en sus características. Como diría el sabio "más sabe el tonto en su casa que el sabio en la ajena".

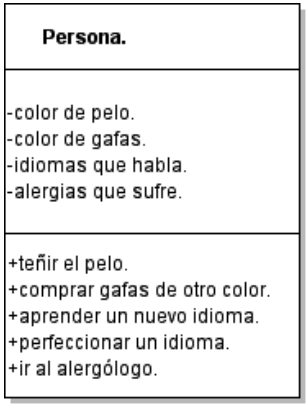
Imaginemos que una persona tiene las siguientes características:

- Color de pelo.
- Color de gafas.
- Idiomas que habla.
- Alergias que sufre.
- Cuantas características más se nos ocurran.

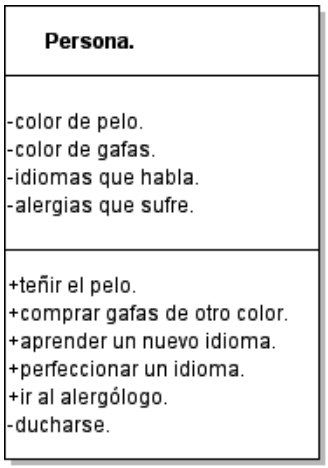
De este modo, cada persona es responsable de *teñirse* el pelo del color que quiera (si es que quiere), es responsable de *comprarse* las gafas de un color u otro, de *aprender más idiomas* o *perfeccionar* los que sabe, de ir al alergólogo o no...

Digamos que en el paradigma estructurado, al no existir el ámbito de privacidad, era fácil que un módulo externo modificase las características privadas de un objeto. En la programación orientada a objetos, un objeto diferente al propietario de las características no debería modificarlas si no es a través de las acciones que el objeto tiene establecidas. En otras palabras, es como si le dijéramos al objeto "debes teñirte el pelo de tal color", siendo el objeto responsable del teñido. O sea, cada objeto es responsable de modificar sus características.

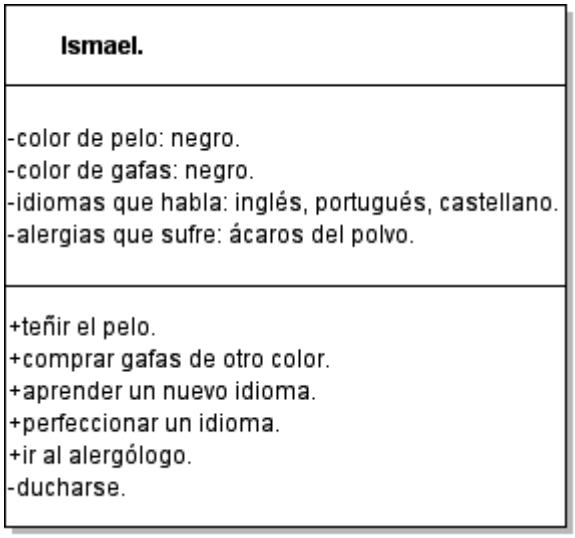
En el ámbito de la POO, la clase anterior se modelaría de la siguiente forma:



Como podemos ver en este ejemplo, las características propias de una clase (molde de un objeto) las señalizamos precedidas de un signo "-", porque son privadas; y las acciones públicas que pueden invocar otros objetos las precedemos de un signo "+". Una clase también puede tener acciones privadas que sólo pueden invocar las acciones propias de la clase. Por ejemplo, podríamos tener la acción "ducharse", invocable desde cualquier acción de las anteriores, pero que sólo puede ejecutar la propia clase.



Lo que hemos visto hasta ahora es un ejemplo de clase con características y acciones. Un objeto no es más que una instancia particular de una clase. Por ejemplo, podríamos tener un objeto de la clase "Persona" llamado *Ismael*. En el lenguaje UML (lenguaje de modelado unificado), quedaría como sigue:



3.2 Atributos y métodos

Hemos visto los dos conceptos fundamentales de POO: clases y objetos. Del mismo modo hemos tenido una aproximación a los atributos y los métodos. En nuestro ejemplo, podemos realizar la siguiente igualdad:

- Características = atributos.
- Acciones = métodos.

Comencemos con el código Java. Los ficheros fuente en Java deben estar dentro de paquetes. Un paquete es una abstracción de un espacio de nombres. Digamos que una clase puede ver todas las clases que hay en su paquete. Para poder ver las clases que hay en otros paquetes debe importar las clases una por una o bien importar el paquete completo en el que está la clase.

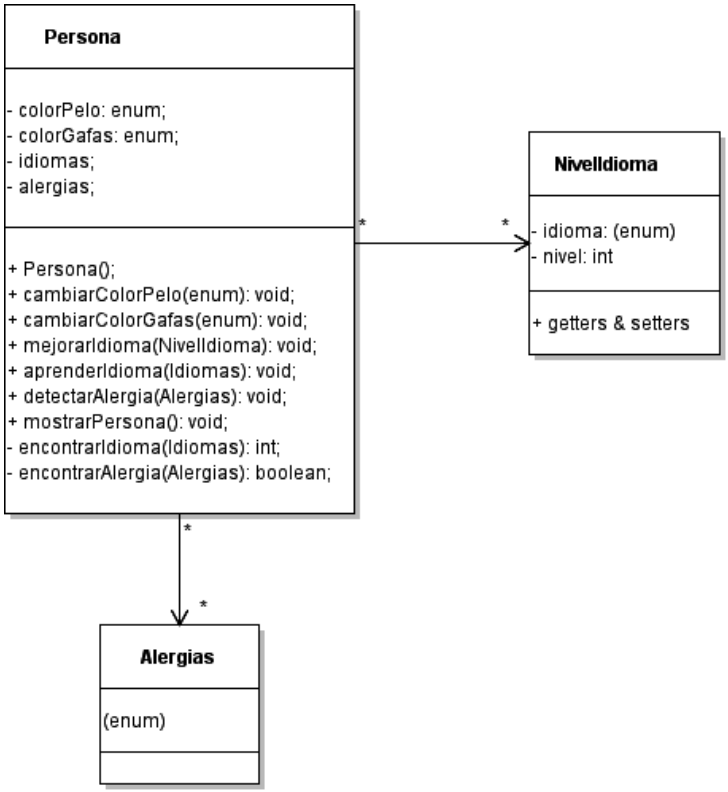
Lo normal es que los atributos de una clase sean privados y que los métodos de la clase sean públicos. Como ya se ha comentado también, pueden existir (de hecho, existen habitualmente) métodos privados de la propia clase.

Existen muchas formas de definir los atributos de las clases. Tantas como tipos de datos simples. Los tipos de datos enumerados se definen de la siguiente forma en Java:

```
package paquete;
public enum ColorPelo {
  NEGRO, RUBIO, PELIRROJO, CASTANO;
}
```

Lo cual, volviendo a lo que hemos explicado, indica que dicho tipo de dato enumerado llamado *ColorPelo* pertenece al paquete llamado *paquete* y que sus posibles valores son negro, rubio, pelirrojo y castaño. Los valores de los tipos enumerados se definen en mayúscula por convención. Veamos el código que define el ejemplo de la clase "Persona". Los valores de los atributos serán enumerados.

El modelo del dominio del ejemplo que vamos a ilustrar en Java es el siguiente:



Del modelo del dominio podemos deducir que la clase "Persona" tendrá dos listas, una de alergias y otra de nivel de idiomas. Veamos su traducción al lenguaje Java. Primero veamos la estructura de los ficheros de código fuente y su distribución en paquetes.

Hemos creado dos paquetes: dominio (con el modelo del dominio) y principal (con el método *main*). Evidentemente, todas las clases y enumerados son visibles dentro del mismo paquete. Para hacer uso de las clases y enumerados en el paquete principal

hemos de usar las correspondientes instrucciones *import*. Veamos los ficheros de código fuente. Empecemos por los enumerados.

- "ColorPelo.java":

```
package dominio;
public enum ColorPelo {
   NEGRO, RUBIO, PELIRROJO, CASTANO;
}
```

- "ColorGafas.java":

```
package dominio;
public enum ColorGafas {
   NEGRO, GRIS, AZUL, ROJO;
}
```

- "Idiomas.java":

```
package dominio;
public enum Idiomas {
   CASTELLANO, INGLES, FRANCES, PORTUGUES, ALEMAN, CHINO;
}
```

- "Alergias.java":

```
package dominio;
public enum Alergias {
   ACAROS, LACTOSA, POLEN, GRAMINEAS;
}
```

- Clase "NivelIdioma" con sus *getters & setters*: "NivelIdioma.java":

```
package dominio;
public class NivelIdioma {
   private Idiomas idioma;
   private int nivel;
   public Idiomas getIdioma() {
      return idioma;
   }
   public void setIdioma(Idiomas idioma) {
      this.idioma = idioma;
   }
   public int getNivel() {
      return nivel;
   }
```

```
    public void setNivel(int nivel) {
        this.nivel = nivel;
    }
}
```

- Clase "Persona.java":

```
package dominio;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Persona {
    private ColorPelo colorPelo;
    private ColorGafas colorGafas;
    private List<NivelIdioma> idiomas;
    private List<Alergias> alergias;
    public Persona() { // Constructor de la clase
        this.colorPelo = ColorPelo.CASTANO;
        this.colorGafas = ColorGafas.NEGRAS;
        this.idiomas = new ArrayList<NivelIdioma>();
        this.alergias = new ArrayList<Alergias>();
    }
    public void cambiarColorPelo(ColorPelo nuevoColor) {
        this.colorPelo = nuevoColor;
    }
    public void cambiarColorGafas(ColorGafas nuevoColor) {
        this.colorGafas = nuevoColor;
    }
    // Prerrequisito: el idioma debe estar en la lista.
    // Si no existe se añadirá como nuevo idioma.
    public void mejorarIdioma(NivelIdioma nivelIdioma) {
        Idiomas idioma = nivelIdioma.getIdioma();
        int lugarLista = this.encontrarIdioma(idioma);
        if (lugarLista == -1) {
            this.idiomas.add(nivelIdioma);
        } else {
            NivelIdioma nivelIdiomaEncontrado
                = this.idiomas.get(lugarLista);
            nivelIdiomaEncontrado.setNivel(nivelIdioma.getNivel());
        }
    }
    // Prerrequisito: el idioma no debe estar en la lista.
    // Si existe no se añadirá.
    public void aprenderIdioma(Idiomas nuevoIdioma) {
        int lugarLista = this.encontrarIdioma(nuevoIdioma);
        if (lugarLista == -1) {
            NivelIdioma nivelIdioma = new NivelIdioma();
```

```
            nivelIdioma.setIdioma(nuevoIdioma);
            nivelIdioma.setNivel(0);
            this.idiomas.add(nivelIdioma);
        }
    }
    // Prerrequisito: la alergia no debe estar en la lista.
    // Si existe en la lista no se añadirá.
    public void detectarAlergia(Alergias nuevaAlergia) {
        if (!this.encontrarAlergia(nuevaAlergia)) {
            this.alergias.add(nuevaAlergia);
        }
    }
    public void mostrarPersona() {
        System.out.println("Los datos de la "
                + "persona son los siguientes.".);
        System.out.print("El color del pelo es: ");
        if (this.colorPelo == ColorPelo.NEGRO) {
            System.out.println("negro".);
        } else if (this.colorPelo == ColorPelo.RUBIO) {
            System.out.println("rubio".);
        } else if (this.colorPelo == ColorPelo.CASTANO) {
            System.out.println("castaño".);
        } else if (this.colorPelo == ColorPelo.PELIRROJO) {
            System.out.println("pelirrojo".);
        }
        System.out.print("El color de sus gafas es: ");
        if (this.colorGafas == ColorGafas.AZUL) {
            System.out.println("azul".);
        } else if (this.colorGafas == ColorGafas.GRIS) {
            System.out.println("gris".);
        } else if (this.colorGafas == ColorGafas.NEGRO) {
            System.out.println("negro".);
        } else if (this.colorGafas == ColorGafas.ROJO) {
            System.out.println("rojo".);
        }
        Iterator it = this.idiomas.iterator();
        if (!it.hasNext()) {
            System.out.println("No habla idiomas".);
        } else {
            System.out.println("Los idiomas que habla son: ");
            while (it.hasNext()) {
                NivelIdioma nivelEsteIdioma = (NivelIdioma) it.next();
                Idiomas idioma = nivelEsteIdioma.getIdioma();
                int nivelEsteIdiomaValor = nivelEsteIdioma.getNivel();
                if (idioma == Idiomas.ALEMAN) {
                    System.out.println("- " + "Alemán "
```

```
                + "- Nivel: " + nivelEsteIdiomaValor + "".);
            } else if (idioma == Idiomas.CASTELLANO) {
                System.out.println("- " + "Castellano "
                + "- Nivel: " + nivelEsteIdiomaValor + "".);
            } else if (idioma == Idiomas.CHINO) {
                System.out.println("- " + "Chino "
                + "- Nivel: " + nivelEsteIdiomaValor + "".);
            } else if (idioma == Idiomas.FRANCES) {
                System.out.println("- " + "Francés "
                + "- Nivel: " + nivelEsteIdiomaValor + "".);
            } else if (idioma == Idiomas.INGLES) {
                System.out.println("- " + "Inglés "
                + "- Nivel: " + nivelEsteIdiomaValor + "".);
            } else if (idioma == Idiomas.PORTUGUES) {
                System.out.println("- " + "Portugués "
                + "- Nivel: " + nivelEsteIdiomaValor + "".);
            }
        }
    }
    it = this.alergias.iterator();
    if (!it.hasNext()) {
        System.out.println("No tiene alergias");
    } else {
        System.out.println("La persona sufre las siguientes alergias:");
        while (it.hasNext()) {
            Alergias estaAlergia = (Alergias) it.next();
            if (estaAlergia == Alergias.ACAROS) {
                System.out.println("- Alérgico a los ácaros del polvo".);
            } else if (estaAlergia == Alergias.GRAMINEAS) {
                System.out.println("- Alérgico a las gramíneas".);
            } else if (estaAlergia == Alergias.LACTOSA) {
                System.out.println("- Intolerante a la lactosa".);
            } else if (estaAlergia == Alergias.POLEN) {
                System.out.println("- Alérgico al polen".);
            }
        }
    }
}
private int encontrarIdioma(Idiomas idioma) {
    int i = 0;
    boolean encontrado = false;
    Iterator it = this.idiomas.iterator();
    while (it.hasNext() && !encontrado) {
        NivelIdioma esteNivelIdioma = (NivelIdioma) it.next();
        Idiomas esteIdioma = esteNivelIdioma.getIdioma();
        if (esteIdioma == idioma) {
```

```
            encontrado = true;
          } else {
            i++;
          }
        }
        if (!encontrado) {
          i = -1;
        }
        return i;
    }
    private boolean encontrarAlergia(Alergias alergia) {
        boolean encontrado = false;
        Iterator it = this.alergias.iterator();
        while (it.hasNext() && !encontrado) {
          Alergias estaAlergia = (Alergias) it.next();
          if (estaAlergia == alergia) {
            encontrado = true;
          }
        }
        return encontrado;
    }
}
```

- Finalmente, la clase principal que hace ejecutar todo el código, "Main.java":

```
package principal;
import dominio.*;
public class Main {
    public static void main(String[] args) {
        Persona ismael = new Persona();
        ismael.cambiarColorPelo(ColorPelo.NEGRO);
        ismael.cambiarColorGafas(ColorGafas.NEGRO);
        ismael.aprenderIdioma(Idiomas.CASTELLANO);
        ismael.aprenderIdioma(Idiomas.INGLES);
        ismael.aprenderIdioma(Idiomas.PORTUGUES);
        NivelIdioma nivelIdiomaCastellano = new NivelIdioma();
        NivelIdioma nivelIdiomaIngles = new NivelIdioma();
        NivelIdioma nivelIdiomaPortugues = new NivelIdioma();
        nivelIdiomaCastellano.setIdioma(Idiomas.CASTELLANO);
        nivelIdiomaCastellano.setNivel(5); // Lengua materna
        nivelIdiomaIngles.setIdioma(Idiomas.INGLES);
        nivelIdiomaIngles.setNivel(3); // Intermedio
        nivelIdiomaPortugues.setIdioma(Idiomas.PORTUGUES);
        nivelIdiomaPortugues.setNivel(2); // Básico avanzado
        ismael.mejorarIdioma(nivelIdiomaIngles);
        ismael.mejorarIdioma(nivelIdiomaCastellano);
```

```
        ismael.mejorarIdioma(nivelIdiomaPortugues);
        ismael.detectarAlergia(Alergias.ACAROS);
        ismael.detectarAlergia(Alergias.ACAROS);
        ismael.mostrarPersona();
    }
}
```

Ejecutando nuestro código, tenemos la siguiente salida en consola:

```
Los datos de la persona son los siguientes...
El color del pelo es: negro.
El color de sus gafas es: negro.
Los idiomas que habla son:
- Castellano - Nivel: 5.
- Inglés - Nivel: 3.
- Portugués - Nivel: 2.
La persona sufre las siguientes alergias:
- Alérgico a los ácaros del polvo.
```

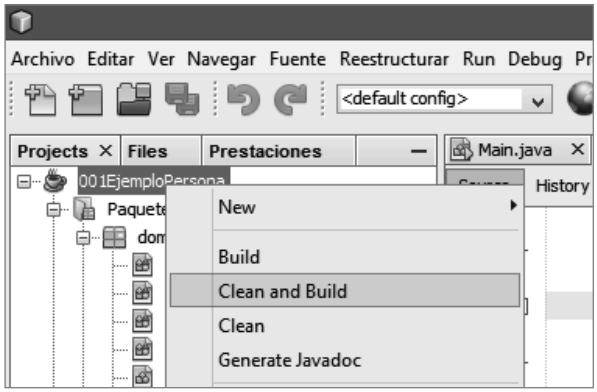
En el ejemplo mostramos el uso de la variable *this* para hacer referencia a los atributos y los métodos de la propia clase. Vemos que la sintaxis del lenguaje Java es similar a la sintaxis del lenguaje C++. Este capítulo pretende dar un somero repaso a la programación orientada a objetos y la programación en Java. El manual no pretende ser un manual de lenguaje Java en sí. Si se tiene dificultad con la comprensión de este ejemplo, el lector debe leer y practicar previamente con la sintaxis del lenguaje Java.

3.3 Método *main*. Archivo Manifest

En el proyecto Java que estemos desarrollando debe haber un método *main* (exceptuando el caso de los *applet*s, donde toma el control el programa "appletviewer.exe"). Si quisiéramos distribuir el programa que hemos escrito en el apartado anterior, lo ideal sería distribuir el programa compilado en *bytecode*. Pero para poder ejecutarlo debemos indicarle a la máquina virtual de Java (JRE) en qué clase se encuentra el método *main* de nuestro proyecto. El método *main* se encuentra en la clase estática "Main", situada en el paquete principal. Debemos indicar dicha información en el archivo Manifest.

El método *main* se encuentra siempre en una clase estática. Una *clase estática* es aquella que no necesita objeto para poder ejecutar sus métodos. En relación a lo que hemos explicado de clases y objetos, digamos que a las clases estáticas "les basta con el molde".

Cualquier IDE de desarrollo en Java permite crear archivos ".jar" compilados con archivos Manifest. Veamos cómo hacerlo en NetBeans. Con el botón derecho sobre el proyecto pulsamos en "Clean and Build".

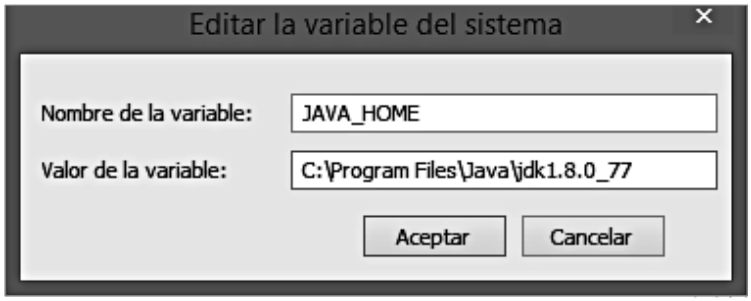


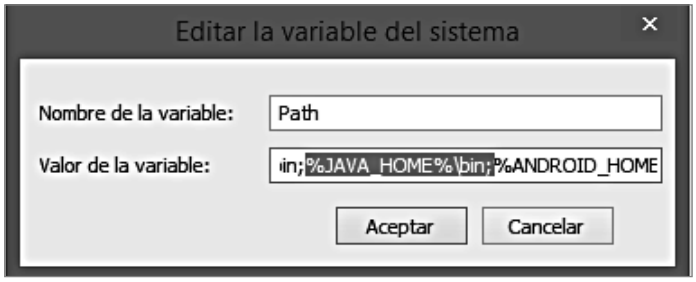
En nuestro *workspace* se habrá creado una carpeta *dist* dentro del proyecto que contendrá el archivo ".jar" compilado.

Dentro de cada una de las carpetas tendremos los archivos ".class" compilados en *bytecode*, mientras que la carpeta "META-INF" lo que contiene es el archivo "MANIFEST.MF". Veamos el contenido del archivo Manifest:

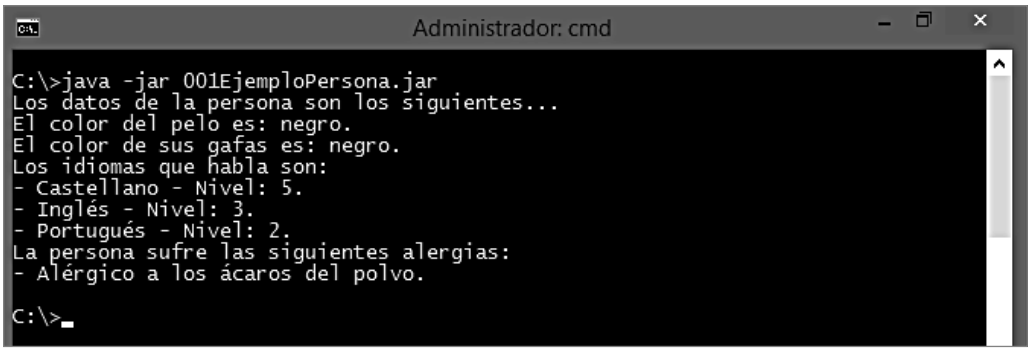
```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.9.4
Created-By: 1.8.0_77-b03 (Oracle Corporation)
Class-Path:
X-COMMENT: Main-Class will be added automatically by build
Main-Class: principal.Main
```

El archivo Manifest sirve para indicarle a la JRE a qué clase debe ir para encontrar el método "public static void main(String[] args){}". De hecho, con el archivo ".jar" generado, podemos distribuir nuestra aplicación sin necesidad de que el usuario tenga instalada en su máquina la JDK. Con la JRE será suficiente. Sí es cierto que en la ruta PATH del sistema operativo debe estar la ruta hasta la entrada a la JRE (fichero "java.exe").





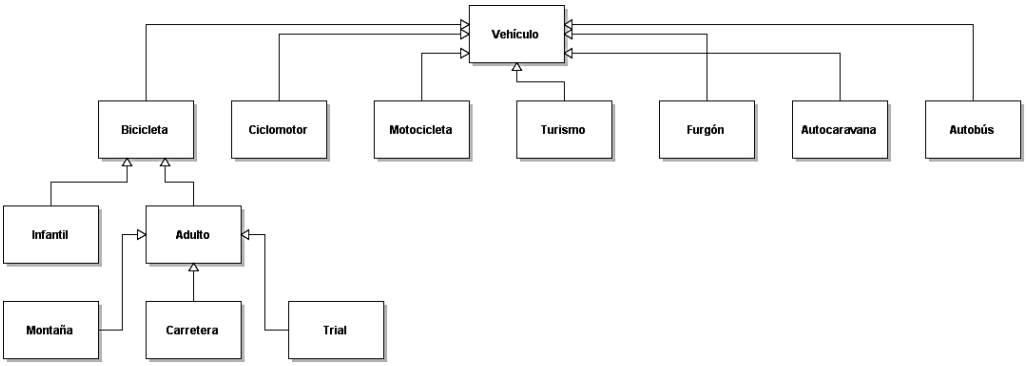
La variable "Path" incluye la variable "JAVA_HOME", y la variable "JAVA_HOME" incluye la ruta hacia "java.exe". Ahora en el navegador podemos escribir: "java -jar 001EjemploPersona.jar".



3.4 Herencia

Si hay una característica que define la taxonomía de los objetos del mundo real, esta es, sin lugar a dudas, la herencia. La taxonomía es la ciencia de la clasificación. Podemos realizar una clasificación del mundo de los vehículos. En la raíz de la clasificación estaría la clase "vehículo". Tenemos varios tipos de vehículos: bicicleta, ciclomotor, motocicleta, turismo, furgón, autocaravana o autobús (y muchos más). Del mismo modo, si definimos cada tipo del subtipo "bicicleta", podemos ver que tenemos

de tipo infantil o adulto. Dentro de las bicicletas de adulto tenemos de tipo montaña, de carretera, para practicar trial... Veamos cómo quedaría definida esta clasificación en el lenguaje de modelado unificado UML.



Evidentemente, todos los vehículos comparten datos. En primera instancia se nos ocurren tres datos que comparten todos los tipos de vehículos:

- Número de ruedas.
- Máxima velocidad que puede alcanzar.
- Máxima masa que puede soportar.

Acudiendo a la "bicicleta", también se nos ocurren características que pueden compartir estos tipos de vehículos.

- Número de velocidades.
- ¿Sirve para competir? Es un dato de tipo Sí/No (booleano).

Acudiendo al subtipo de "adulto", podemos tener otro dato booleano:

- ¿*Amateur* o profesional? Es un dato de tipo Sí/No (booleano). Sí: profesional. No: *amateur*.

La idea de la herencia es que cada subtipo de objeto tenga todas las características, tan públicas como protegidas de sus ancestros. Por ejemplo, una bicicleta de adulto de carretera tendrá todas las características siguientes:

- Número de ruedas.
- Máxima velocidad que puede alcanzar.
- Máxima masa que puede soportar.
- Número de velocidades.
- ¿Sirve para competir?
- ¿*Amateur* o profesional?

Como hemos indicado, los atributos o métodos protegidos son aquellos que son visibles por las clases que heredan de otra, pero no por clases ajenas. La palabra reservada con la que definimos estos atributos es *protected*.

La palabra reservada con la que definimos una clase que deriva o extiende de otra es *extends*.

Como ejemplo, definamos las clases "Vehiculo", "Bicicleta", "BicicletaAdulto", "BicicletaInfantil" y "BicicletaCarretera", así como una clase principal que haga uso de ellas.

Veamos cómo estructuramos el código en paquetes dentro de nuestro nuevo proyecto Java.

- Clase "Vehiculo.java":

```
package dominio;
public class Vehiculo {
   protected int numeroRuedas;
   protected int masaMaximaSoportable; // Kg
   protected int velocidadMaximaAlcanzable; // Km/hora
   public int getNumeroRuedas() {
      return numeroRuedas;
   }
   public void setNumeroRuedas(int numeroRuedas) {
      this.numeroRuedas = numeroRuedas;
   }
   public int getMasaMaximaSoportable() {
      return masaMaximaSoportable;
   }
   public void setMasaMaximaSoportable(int masaMaximaSoportable) {
      this.masaMaximaSoportable = masaMaximaSoportable;
   }
```

```
  public int getVelocidadMaximaAlcanzable() {
    return velocidadMaximaAlcanzable;
  }
  public void setVeloci dadMaximaAlcanzable(int velocidadMaximaAlcanzable) {
    this.velocidadMaximaAlcanzable = velocidadMaximaAlcanzable;
  }
}
```

- Clase "Bicicleta.java":

```
package dominio;

public class Bicicleta extends Vehiculo {
  protected int numeroVelocidades;
  protected boolean sirveParaCompetir;
  public int getNumeroVelocidades() {
    return numeroVelocidades;
  }
  public void setNumeroVelocidades(int numeroVelocidades) {
    this.numeroVelocidades = numeroVelocidades;
  }
  public boolean isSirveParaCompetir() {
    return sirveParaCompetir;
  }
  public void setSirveParaCompetir(boolean sirveParaCompetir) {
    this.sirveParaCompetir = sirveParaCompetir;
  }
}
```

- Clase "BicicletaInfantil.java":

```
package dominio;
public class BicicletaInfantil extends Bicicleta {}
```

- Clase "BicicletaAdulto.java":

```
package dominio;
public class BicicletaAdulto extends Bicicleta {
  protected boolean profesionalAmateur; // true = profesional
  public boolean isProfesionalAmateur() {
    return profesionalAmateur;
  }
  public void setProfesionalAmateur(boolean profesionalAmateur) {
    this.profesionalAmateur = profesionalAmateur;
  }
}
```

- Clase "BicicletaCarretera.java":

```
package dominio;
public class BicicletaCarretera extends BicicletaAdulto {}
```

- Clase principal. Creamos una instancia concreta de bicicleta de carretera perteneciente a Ismael:

```
package principal;
import dominio.BicicletaCarretera;
public class Main {
  public static void main(String[] args) {
    // Definiéndolo de tipo BicicletaCarretera podemos ver que
    // tiene todos los métodos públicos de sus ancestros.
    BicicletaCarretera miBicicletaCarretera = new BicicletaCarretera();
    miBicicletaCarretera.setVelocidadMaximaAlcanzable(90); // km/hora.
    miBicicletaCarretera.setMasaMaximaSoportable(160); // kg.
    miBicicletaCarretera.setNumeroRuedas(2);
    miBicicletaCarretera.setNumeroVelocidades(12); // 2 platos y 6 piñones.
    miBicicletaCarretera.setSirveParaCompetir(false); // No es de competir.
    miBicicletaCarretera.setProfesionalAmateur(false); // Es de aficionado.
    // Mostramos los datos por consola.
    System.out.println("Datos de la bicicleta de Ismael".);
    System.out.println("La velocidad máxima alcanzable es " +
      miBicicletaCarretera.getVelocidadMaximaAlcanzable() + " km/hora".);
    System.out.println("La masa máxima soportable es " +
      miBicicletaCarretera.getMasaMaximaSoportable() + " kg".);
    System.out.println("El número de ruedas es " +
      miBicicletaCarretera.getNumeroRuedas()+ "".);
    System.out.println("El número de velocidades de la bicicleta es " +
      miBicicletaCarretera.getNumeroVelocidades() + "".);
    if(miBicicletaCarretera.isSirveParaCompetir()) {
      System.out.println("Sirve para competir".);
    } else {
      System.out.println("No sirve para competir".);
    }
    if(miBicicletaCarretera.isProfesionalAmateur()) {
      System.out.println("Es bicicleta profesional".);
    } else {
      System.out.println("Es bicicleta de aficionado".);
    }
  }
}
```

La salida por consola de la aplicación es la siguiente:

Datos de la bicicleta de Ismael.
La velocidad máxima alcanzable es 90 km/hora.
La masa máxima soportable es 160 kg.
El número de ruedas es 2.
El número de velocidades de la bicicleta es 12.
No sirve para competir.
Es bicicleta de aficionado.

En el ejemplo, hemos podido comprobar como todos los métodos públicos de los ancestros de la clase "BicicletaCarretera" están disponibles. Los atributos también están disponibles porque los hemos definido como *protected* o *protegidos*.

La variable "miBicicletaCarretera" se ha definido como de tipo "BicicletaCarretera". Si la hubiéramos definido de una superclase superior, sólo podríamos acceder a los métodos definidos hasta esa superclase superior. Por ejemplo, redefinamos la clase principal definiendo "miBicicletaCarretera" de tipo "Vehiculo":

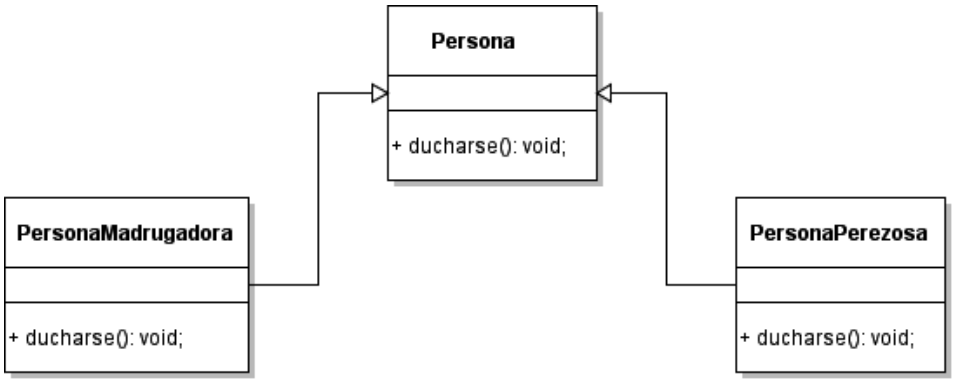
```
package principal;
import dominio.BicicletaCarretera;
import dominio.Vehiculo;
public class Main {
  public static void main(String[] args) {
    // Definiéndolo de tipo Vehículo vemos que sólo tiene
    // los métodos públicos de "Vehiculo".
    Vehiculo miBicicletaCarretera = new BicicletaCarretera();
    miBicicletaCarretera.setVelocidadMaximaAlcanzable(90); // Km/hora.
    miBicicletaCarretera.setMasaMaximaSoportable(160); // Kg.
    miBicicletaCarretera.setNumeroRuedas(2);
    // Mostramos los datos por consola.
    System.out.println("Datos de la bicicleta de Ismael".);
    System.out.println("La velocidad máxima alcanzable es " +
      miBicicletaCarretera.getVelocidadMaximaAlcanzable() + " km/hora".);
    System.out.println("La masa máxima soportable es " +
      miBicicletaCarretera.getMasaMaximaSoportable() + " kg".);
    System.out.println("El número de ruedas es " +
      miBicicletaCarretera.getNumeroRuedas()+ "".);
  }
}
```

No tenemos acceso a los métodos definidos ni en la clase "Bicicleta" ni en la clase "BicicletaAdulto". La salida por consola es la siguiente:

Datos de la bicicleta de Ismael.
La velocidad máxima alcanzable es 90 km/hora.
La masa máxima soportable es 160 kg.
El número de ruedas es 2.

3.5 Polimorfismo

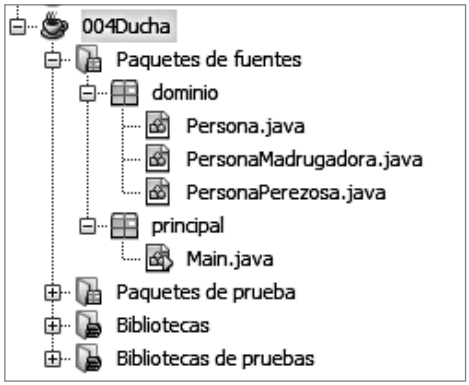
El *polimorfismo* se define como la "cualidad de polimorfo" (RAE). Buscando *polimorfo* en el diccionario de la RAE nos encontramos que es un adjetivo que define aquello "que tiene o puede tener distintas formas". En el contexto de la programación (programación orientada a objetos), se define como aquel código que, aun escrito igual, puede lanzar ejecuciones diferentes de código. Esto lo conseguimos gracias a la herencia que hemos estudiado en el apartado previo. Veamos un ejemplo para entenderlo más fácilmente. Tenemos la siguiente estructura de clases:



En el ejemplo podemos ver como una superclase define un método público denominado *ducharse*. Digamos que las personas normales crearían instancias (objetos) de dicha clase. Dichas personas se duchan a las ocho de la mañana, antes de ir al trabajo. Pero tenemos una clasificación. Tenemos dos tipos de personas: personas madrugadoras y personas perezosas. Las personas madrugadoras se levantan muy temprano y se duchan a las siete de la mañana. Y las personas perezosas se van al trabajo sin ducharse y se duchan por la tarde cuando les apetece.

En nuestro código Java, cuando creemos una persona, definiremos el tipo en la creación. A partir de este punto, el código se ejecutará dependiendo de la instancia que

hayamos creado. Veamos este ejemplo en forma de código Java. Nuestra estructura de ficheros fuente es la siguiente:



- Clase "Persona.java":

```
package dominio;
public class Persona {
   public void ducharse() {
      System.out.println("Soy una persona normal y me ducho a las ocho"
            + " antes de ir al trabajo".);
   }
}
```

- Clase "PersonaMadrugadora.java":

```
package dominio;
public class PersonaMadrugadora extends Persona {
   @Override
   public void ducharse() {
      System.out.println("Soy una persona madrugadora y como me despierto "
            + "temprano me aburro y me ducho a las siete".);
   }
}
```

- Clase "PersonaPerezosa.java":

```
package dominio;
public class PersonaPerezosa extends Persona {
   @Override
   public void ducharse() {
      System.out.println("Soy una persona perezosa. Me levanto tarde "
            + "y me ducho cuando puedo".);
   }
}
```

Y ahora el programa principal. Vamos a crear una persona normal:

```
package principal;
import dominio.Persona;
public class Main {
   public static void main(String[] args) {
      Persona p = new Persona();
      p.ducharse();
   }
}
```

La salida por consola de esta aplicación es la siguiente:

Soy una persona normal y me ducho a las ocho antes de ir al trabajo.

En el anterior código hay una instrucción, en concreto "p.ducharse();", que puede tomar múltiples formas, dependiendo del objeto al que hayamos enlazado dinámicamente la referencia de "p" de la clase "Persona". Veamos otra versión del programa principal:

```
package principal;
import dominio.Persona;
import dominio.PersonaMadrugadora;
public class Main {
   public static void main(String[] args) {
      Persona p = new PersonaMadrugadora();
      p.ducharse();
   }
}
```

El "enlace dinámico" se realiza ahora a otro código. Veamos la salida por consola:

Soy una persona madrugadora y como me despierto temprano me aburro y me ducho a las siete.

Veamos, por último, el polimorfismo de la citada instrucción con el método de la clase "PersonaPerezosa":

```
package principal;
import dominio.Persona;
import dominio.PersonaPerezosa;
public class Main {
   public static void main(String[] args) {
      Persona p = new PersonaPerezosa();
      p.ducharse();
   }
}
```

La salida por consola es la siguiente:

Soy una persona perezosa. Me levanto tarde y me ducho cuando puedo.

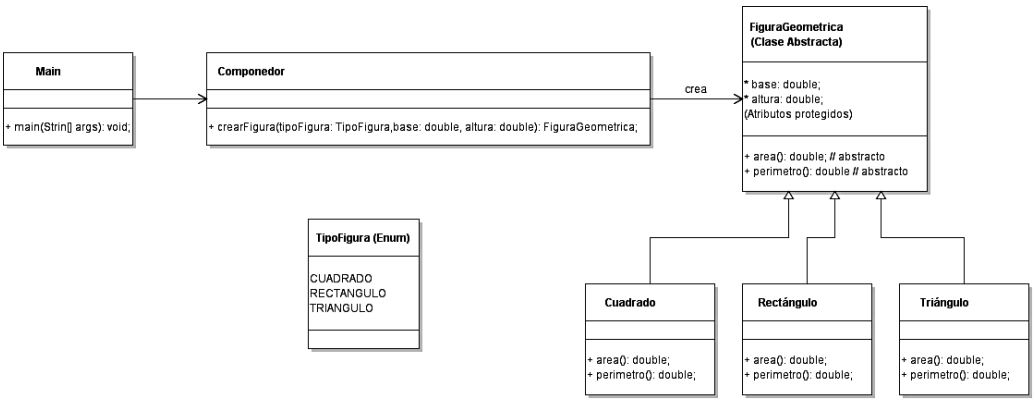
3.6 Clases abstractas

Una clase abstracta es aquella de la que nunca se va a crear ninguna instancia. Sin embargo, se crearán instancias de sus clases heredadas. Existen clases en las que no tiene sentido crear una instancia concreta (por ser conceptos abstractos, por no requerirse en la funcionalidad de la aplicación..., por las razones que se nos ocurran).

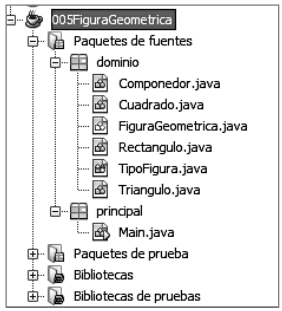
Las clases abstractas también pueden contener métodos abstractos. Son aquellos métodos que deben poseer e implementar las clases que hereden de la clase abstracta. Pero en la clase abstracta no damos la definición del método. Únicamente ofrecemos la cabecera del método que deberán implementar forzosamente las clases derivadas.

Aclaremos todos estos conceptos con un ejemplo. Pensemos en las figuras geométricas. Si creamos la clase "FiguraGeometrica", podemos proteger datos (atributos) comunes a varios tipos de figuras. Del mismo modo podemos definir la cabecera de métodos implementables por varios tipos de figuras geométricas. Pero únicamente definir la cabecera. Evidentemente la clase "FiguraGeometrica" será una clase abstracta. Los métodos cuya cabecera se defina en dicha clase abstracta serán métodos abstractos.

En este ejemplo vamos a hacer uso de dos patrones de diseño orientados a objetos. Uno va a ser el método fábrica (un método cuya función es crear objetos por nosotros, sin tener que usar el cliente la instrucción *new*). El otro patrón será el patrón "Estrategia" o "Strategy". En la que un componedor hará uso del polimorfismo para crear un enlace dinámico a un tipo de objeto. En este caso unificaremos los patrones en el mismo componedor. Veamos el diagrama de clases.



La estructura de ficheros queda como sigue:



Veamos uno por uno los artefactos *software* de este proyecto:

- Enumerado "TipoFigura.java":

```
package dominio;
public enum TipoFigura {
   CUADRADO,
   RECTANGULO,
   TRIANGULO
}
```

- Clase "Componedor.java":

```
package dominio;
public class Componedor {
   public FiguraGeometrica crearFigura(TipoFigura tipoFigura,
         double base, double altura) {
      FiguraGeometrica fg = null;
      if (tipoFigura == TipoFigura.CUADRADO) {
         fg = new Cuadrado(base,altura);
      } else if (tipoFigura == TipoFigura.RECTANGULO) {
         fg = new Rectangulo(base,altura);
      } else if (tipoFigura == TipoFigura.TRIANGULO) {
         fg = new Triangulo(base,altura);
      }
      return fg;
   }
}
```

- Clase "FiguraGeometrica.java" (clase abstracta):

```
package dominio;
public abstract class FiguraGeometrica {
   protected double base;
   protected double altura;
```

```
   public abstract double area();
   public abstract double perimetro();
}
```

- Clase "Cuadrado.java":

```
package dominio
public class Cuadrado extends FiguraGeometrica {
   public Cuadrado(double base, double altura) {
      this.base = base;
      this.altura = altura;
   }
   @Override
   public double area() {
      return (double) this.base * this.altura;
   }
   @Override
   public double perimetro() {
      return (double) (2 * this.base) +  (2 * this.altura);
   }
}
```

- Clase "Rectangulo.java":

```
package dominio;
public class Rectangulo extends FiguraGeometrica {
   public Rectangulo(double base, double altura) {
      this.base = base;
      this.altura = altura;
   }
   @Override
   public double area() {
      return (double) this.base * this.altura;
   }
   @Override
   public double perimetro() {
      return (double) (2 * this.base) +  (2 * this.altura);
   }
}
```

- Clase "Triangulo.java":

```
package dominio;
public class Triangulo extends FiguraGeometrica {
   public Triangulo(double base, double altura) {
```

```
        this.base = base;
        this.altura = altura;
    }
    @Override
    public double area() {
        return (double) (this.base * this.altura) / 2;
    }
    @Override
    public double perimetro() {
        // Suponemos triángulo equilátero.
        return (double) (3 * this.base);
    }
}
```

- Finalmente, vemos la clase principal "Main":

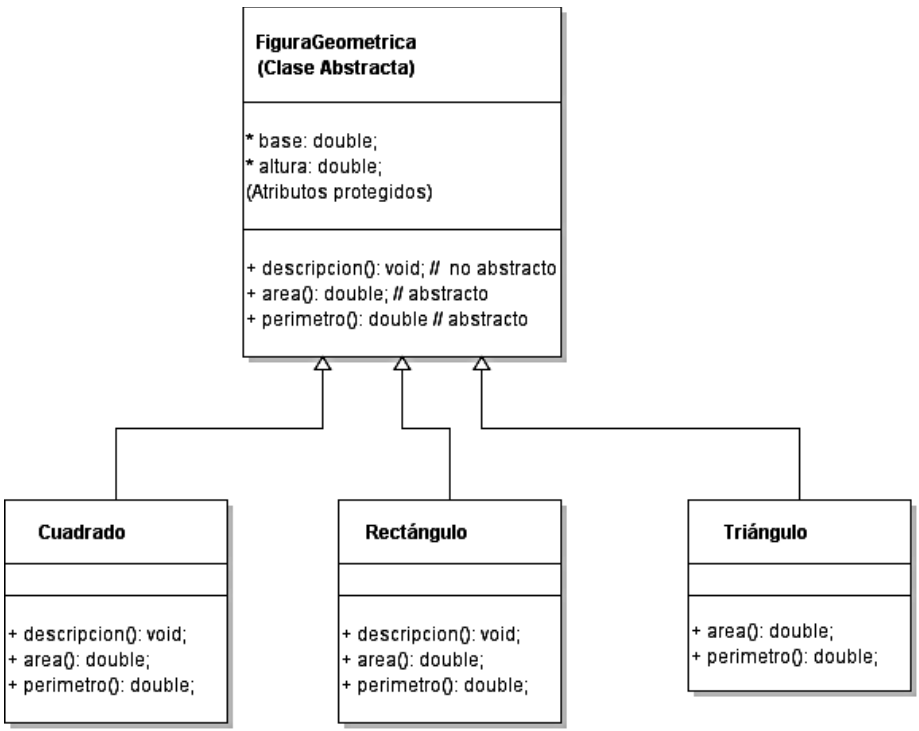
```
package principal;
import dominio.Componedor;
import dominio.FiguraGeometrica;
import dominio.TipoFigura;
public class Main {
    public static void main(String[] args) {
        Componedor c = new Componedor();
        FiguraGeometrica fg = c.crearFigura(TipoFigura.CUADRADO, 3, 3);
        double area = fg.area();
        double perimetro = fg.perimetro();
        System.out.println("Datos del cuadrado".);
        System.out.println("El área es: "+area+"".);
        System.out.println("El perímetro es: "+perimetro+"".);
        fg = c.crearFigura(TipoFigura.RECTANGULO, 5, 3);
        area = fg.area();
        perimetro = fg.perimetro();
        System.out.println("Datos del rectángulo".);
        System.out.println("El área es: "+area+"".);
        System.out.println("El perímetro es: "+perimetro+"".);
        fg = c.crearFigura(TipoFigura.TRIANGULO, 4, 3);
        area = fg.area();
        perimetro = fg.perimetro();
        System.out.println("Datos del triángulo".);
        System.out.println("El área es: "+area+"".);
        System.out.println("El perímetro es: "+perimetro+"".);
    }
}
```

La salida por consola del programa es la siguiente:

Datos del cuadrado.
El área es: 9,0.
El perímetro es: 12,0.
Datos del rectángulo.
El área es: 15,0.
El perímetro es: 16,0.
Datos del triángulo.
El área es: 6,0.
El perímetro es: 12,0.

3.6.1 Métodos implementados en clases abstractas

En cualquier clase abstracta podemos dar una implementación de cualquier método que puede ser sobrescrito en una clase heredada. Retomemos el ejemplo de la figura geométrica. Podemos añadir un método público llamado *descripción* que indique el tipo de figura geométrica sobre el que estamos trabajando.



Vemos que en la clase abstracta podemos implementar el método descripción, que puede ser sobrescrito en las clases heredadas (tal cual hemos dicho). En este ejemplo, la clase "Triangulo" no sobrescribe el método descripción.

Veamos la implementación de las clases:

- Clase "FiguraGeometrica.java":

```
package dominio;
public abstract class FiguraGeometrica {
   protected double base;
   protected double altura;
   public void descripcion() {
      System.out.println("Soy una figura geométrica cualquiera".);
   }
   public abstract double area();
   public abstract double perimetro();
}
```

- Clase "Cuadrado.java":

```
package dominio;
public class Cuadrado extends FiguraGeometrica {
   public Cuadrado(double base, double altura) {
      this.base = base;
      this.altura = altura;
   }
   @Override
   public void descripcion() {
      System.out.println("Soy un cuadrado".);
   }
   @Override
   public double area() {
      return (double) this.base * this.altura;
   }
   @Override
   public double perimetro() {
      return (double) (2 * this.base) +  (2 * this.altura);
   }
}
```

- Clase "Rectangulo.java":

```
package dominio;
public class Rectangulo extends FiguraGeometrica {
   public Rectangulo(double base, double altura) {
      this.base = base;
      this.altura = altura;
   }
```

```
    @Override
    public void descripcion() {
        System.out.println("Soy un rectángulo".);
    }
    @Override
    public double area() {
        return (double) this.base * this.altura;
    }
    @Override
    public double perimetro() {
        return (double) (2 * this.base) +  (2 * this.altura);
    }
}
```

- La clase "Triangulo" permanece igual que la mostrada en el apartado previo.

Como podemos ver, si ejecutamos el método descripción sobre un objeto de la clase "Triangulo", la implementación que se ejecutará será aquella de la superclase "Figura-Geometrica". Veamos la implementación de la clase principal y la salida por consola de la aplicación:

```
package principal;
import dominio.Componedor;
import dominio.FiguraGeometrica;
import dominio.TipoFigura;
public class Main {
    public static void main(String[] args) {
        Componedor c = new Componedor();
        FiguraGeometrica fg = c.crearFigura(TipoFigura.CUADRADO, 3, 3);
        double area = fg.area();
        double perimetro = fg.perimetro();
        fg.descripcion();
        System.out.println("El área es: "+area+"".);
        System.out.println("El perímetro es: "+perimetro+"".);
        fg = c.crearFigura(TipoFigura.RECTANGULO, 5, 3);
        area = fg.area();
        perimetro = fg.perimetro();
        fg.descripcion();
        System.out.println("El área es: "+area+"".);
        System.out.println("El perímetro es: "+perimetro+"".);
        fg = c.crearFigura(TipoFigura.TRIANGULO, 4, 3);
        area = fg.area();
        perimetro = fg.perimetro();
        fg.descripcion();
        System.out.println("El área es: "+area+"".);
        System.out.println("El perímetro es: "+perimetro+"".);
    }
}
```

Ahora, cada tipo de objeto indica de qué clase es. El triángulo debe decir que es una figura geométrica cualquiera. Veamos la salida por consola:

Soy un cuadrado.
El área es: 9,0.
El perímetro es: 12,0.
Soy un rectángulo.
El área es: 15,0.
El perímetro es: 16,0.
Soy una figura geométrica cualquiera.
El área es: 6,0.
El perímetro es: 12,0.

3.7 Clases estáticas

Las clases estáticas son aquellas que no necesitan tener una instancia en memoria (un objeto) para poder hacer uso de sus métodos o atributos. Suelen ser clases sin estado en las que tendremos "métodos funcionalidad" (similares a los de la programación estructurada), constantes, enumerados... Siempre sin estado de los objetos o con estado compartido para toda la aplicación. El ejemplo más conocido es el de la clase "Main" en la que se ubica el método "public static void main(String[] args)".

Es bueno señalar que el modificador *static* no se indica a nivel de clase, sino a nivel de los métodos o atributos necesarios. Veamos un ejemplo. ¿Conocemos la función factorial de un número? La *función factorial de un número* se define como una función recursiva para los enteros positivos (números naturales). Lo calculamos como el propio número multiplicado por el factorial del entero inmediatamente menor. La recursión acaba en el cero. El factorial de cero es uno. Veamos esta explicación matemáticamente:

$$n! = n * (n-1)!$$
$$0! = 1$$

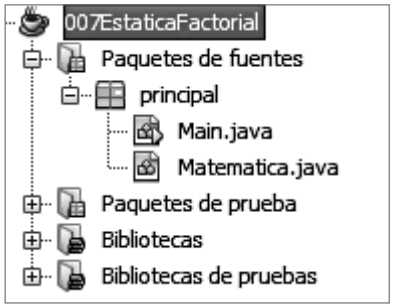
Siguiendo la anterior definición, podemos calcular el factorial de cualquier número entero positivo. Estudiemos el factorial del número 4.

$$4! = 4 * 3!$$
$$3! = 3 * 2!$$
$$2! = 2 * 1!$$
$$1! = 1 * 0!$$
$$0! = 1$$

Por lo tanto, el factorial de 4 es:

$$4! = 4 * 3 * 2 * 1 * 1 = 24$$

Estudiemos una clase estática que implementa la función factorial. Como hemos dicho, será una clase sin estado.



- Clase estática "Matematica.java":

```
package principal;
public class Matematica {
   public static long factorial(long n) {
      if (n>0) {
         return n * (Matematica.factorial(n-1));
      } else if (n == 0) {
         return 1;
      } else {
         return 1;
      }
   }
}
```

- Clase principal "Main.java" (que es otra clase estática):

```
package principal;
public class Main {
   public static void main(String[] args) {
      System.out.println("Vamos a estudiar el factorial de 4".);
      long resultado = Matematica.factorial(4);
      System.out.println("El factorial es: " + resultado + "".);
   }
}
```

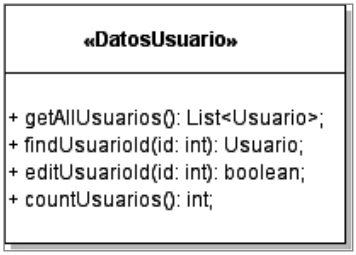
Observamos que no es necesario realizar *new* de ningún objeto. El método factorial es un método de clase y podemos llamarlo invocando únicamente a la clase (clase estática). El resultado por consola es el siguiente:

Vamos a estudiar el factorial de 4.
El factorial es: 24.

3.8 Implementación de interfaces

Cuando en POO nos encontramos con un caso de herencia, dándose la situación de que sólo necesitamos heredar métodos (no atributos) y no necesitamos implementaciones que puedan reutilizarse en las clases heredadas, lo que precisamos, en vez de heredar clases, es implementar interfaces.

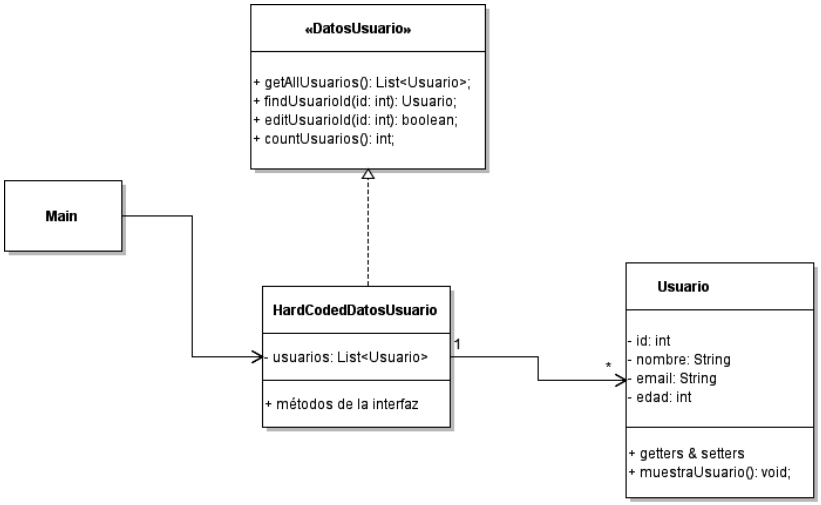
Las interfaces únicamente exponen al público las cabeceras de los métodos que deben ser implementados por las clases que "implementen las interfaces". Veamos un ejemplo de un servicio de acceso a datos de usuarios. La interfaz definirá únicamente los métodos que deben implementar aquellas clases que quieran implementar la interfaz. En este caso definiremos la interfaz "DatosUsuario".



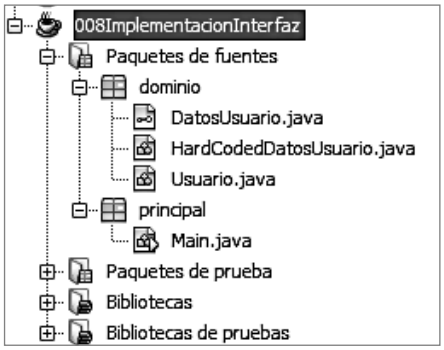
Se define una interfaz con cuatro métodos de acceso a datos. Cualquier clase que quiera implementar dicha interfaz debe implementar dichos métodos.

Para mostrar un ejemplo, imaginemos un acceso a datos *hard coded*, es decir, los datos estarán en un fichero de código fuente.

Crearemos la siguiente estructura de clases:



Para implementar una interfaz debemos usar la palabra reservada *implements*.



- Clase "Usuario.java":

```
package dominio;
public class Usuario {
   private int id;
   private String nombre;
   private String email;
   private int edad;
   public int getId() {
      return id;
   }
   public void setId(int id) {
      this.id = id;
   }
   public String getNombre() {
      return nombre;
   }
   public void setNombre(String nombre) {
      this.nombre = nombre;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }
   public int getEdad() {
      return edad;
   }
   public void setEdad(int edad) {
      this.edad = edad;
   }
```

```
  public void muestraUsuario() {
    System.out.println("Id: "+this.id+"".);
    System.out.println("Nombre: "+this.nombre+"".);
    System.out.println("Email: "+this.email+"".);
    System.out.println("Edad: "+this.edad+"".);
  }
}
```

- Interfaz "DatosUsuario.java":

```
package dominio;
import java.util.List;
public interface DatosUsuario {
  public List<Usuario> getAllUsuarios();
  public Usuario findUsuario(int id);
  public boolean editUsuario(Usuario u);
  public int countUsuarios();
}
```

- Clase "HardCodedDatosUsuario.java":

```
package dominio;
import java.util.ArrayList;
import java.util.List;
public class HardCodedDatosUsuario implements DatosUsuario {
  private List<Usuario> usuarios;
  public HardCodedDatosUsuario() {
    this.usuarios = new ArrayList<Usuario>();
    Usuario u = new Usuario();
    u.setId(1);
    u.setNombre("Ismael");
    u.setEmail("ismael@ismael.com");
    u.setEdad(33);
    this.usuarios.add(u);
    u = new Usuario();
    u.setId(2);
    u.setNombre("Antonio");
    u.setEmail("antonio@antonio.com");
    u.setEdad(44);
    this.usuarios.add(u);
    u = new Usuario();
    u.setId(3);
    u.setNombre("Luis");
    u.setEmail("luis@luis.com");
    u.setEdad(22);
    this.usuarios.add(u);
  }
```

```
    @Override
    public List<Usuario> getAllUsuarios() {
        return this.usuarios;
    }
    @Override
    public Usuario findUsuario(int id) {
        for (Usuario u: this.usuarios) {
            if (id == u.getId()) {
                return u;
            }
        }
        return null;
    }
    @Override
    public boolean editUsuario(Usuario u) {
        for (Usuario us: this.usuarios) {
            if (u.getId() == us.getId()) {
                us.setNombre(u.getNombre());
                us.setEmail(u.getEmail());
                us.setEdad(u.getEdad());
                return true;
            }
        }
        return false;
    }
    @Override
    public int countUsuarios() {
        return this.usuarios.size();
    }
}
```

- Clase "Main.java":

```
package principal;
import dominio.DatosUsuario;
import dominio.HardCodedDatosUsuario;
import dominio.Usuario;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        DatosUsuario du = new HardCodedDatosUsuario();
        int numeroUsuarios = du.countUsuarios();
        System.out.println("El número de usuarios es: "+numeroUsuarios+"".);
        System.out.println("*****Mostramos todos los usuarios*****");
        List<Usuario> usuarios = du.getAllUsuarios();
        for (Usuario u: usuarios) {
```

```
            System.out.println("************************************");
            u.muestraUsuario();
        }
        System.out.println("*****El usuario con id 2 es*****");
        Usuario u = du.findUsuario(2);
        u.muestraUsuario();
        System.out.println("*****Modificamos el usuario con id 2*****");
        Usuario modificado = new Usuario();
        modificado.setId(2);
        modificado.setNombre("Raúl");
        modificado.setEmail("raul@raul.com");
        modificado.setEdad(20);
        if (du.editUsuario(modificado)) {
            System.out.println("*****Mostramos todos los usuarios*****");
            usuarios = du.getAllUsuarios();
            for (Usuario us: usuarios) {
                System.out.println("************************************");
                us.muestraUsuario();
            }
        }
    }
}
```

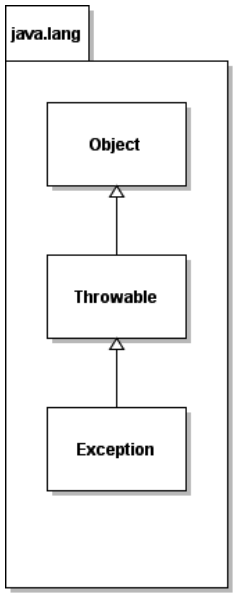
La salida por consola del programa es la siguiente:

```
El número de usuarios es: 3.
*****Mostramos todos los usuarios*****
************************************
Id: 1.
Nombre: Ismael.
Email: ismael@ismael.com.
Edad: 33.
************************************
Id: 2.
Nombre: Antonio.
Email: antonio@antonio.com.
Edad: 44.
************************************
Id: 3.
Nombre: Luis.
Email: luis@luis.com.
Edad: 22.
*****El usuario con id 2 es*****
Id: 2.
Nombre: Antonio.
Email: antonio@antonio.com.
Edad: 44.
```

```
*****Modificamos el usuario con id 2*****
*****Mostramos todos los usuarios*****
************************************
Id: 1.
Nombre: Ismael.
Email: ismael@ismael.com.
Edad: 33.
************************************
Id: 2.
Nombre: Raúl.
Email: raul@raul.com.
Edad: 20.
************************************
Id: 3.
Nombre: Luis.
Email: luis@luis.com.
Edad: 22.
```

3.9 Excepciones

El control de los errores se lleva a cabo en el lenguaje Java a través de las excepciones. Las excepciones son un tipo especial de objeto. Al ser un tipo especial de objeto, hereda de la clase "Object", que se encuentra en el paquete "java.lang". La clase "Object" es aquella de la cual heredan todas las clases que existen en la plataforma Java, así como todas aquellas clases que creemos en el desarrollo de nuestras aplicaciones. La jerarquía de clases de la clase "Exception" es la siguiente:



En nuestras aplicaciones, aparte de las excepciones que proporcione la plataforma Java, crearemos nuestras propias excepciones. Una excepción típica es aquella de acceso a datos en nuestra base de datos. Otro tipo de excepciones son las excepciones lógicas.

Empecemos por lo simple y veamos cómo tratamos las excepciones de la propia plataforma Java JSE. Veamos un pequeño programa que realiza una división entre cero:

```
package principal;
public class Main {
   public static void main(String[] args) {
      int d = 9;
      int result = d / 0;
      System.out.println("El resultado es "+result+"".);
   }
}
```

Al ejecutar el anterior código, la salida que tenemos por consola es la siguiente:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at principal.Main.main(Main.java:7)
```

Vemos que la excepción que se ha lanzado es "ArithmeticException", que hereda de la clase "Exception" vista anteriormente. La forma de evitar este tipo de errores que se producen en tiempo real (normalmente usaremos para dividir variables en vez de valores escritos en el código) es mediante el uso de las instrucciones "try... catch". Veamos cómo se hace:

```
package principal;
public class Main {
   public static void main(String[] args) {
      int d = 9;
      try {
         int result = d / 0;
         System.out.println("El resultado es "+result+"".);
      } catch(ArithmeticException e) {
         System.out.println("Error: "+e.getMessage()+"".);
      }
   }
}
```

Ahora la salida por consola será la siguiente:

Error: / by zero.

La lógica es: situamos en un bloque "try" aquellas instrucciones susceptibles de producir error o que deben ejecutarse secuencialmente si no se produce el error. En el bloque "catch" hacemos indicación de la excepción que vamos a capturar, de cuyos

métodos podremos hacer uso. Uno de dichos métodos es "getMessage()", que muestra la causa del error. Si en vez de entre cero dividimos entre tres:

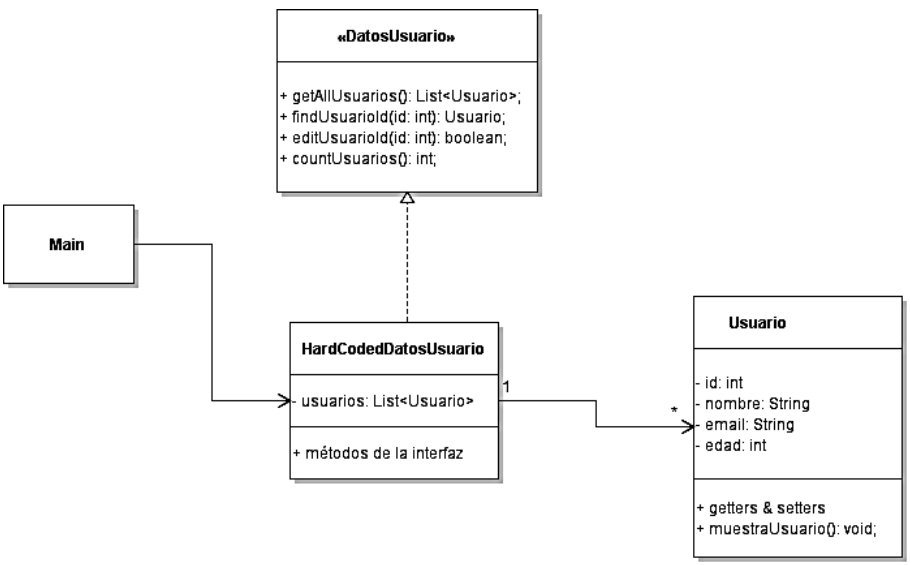
```
package principal;
public class Main {
   public static void main(String[] args) {
      int d = 9;
      try {
         int result = d / 3;
         System.out.println("El resultado es "+result+"".);
      } catch(ArithmeticException e) {
         System.out.println("Error: "+e.getMessage()+"".);
      }
   }
}
```

La salida por consola es:

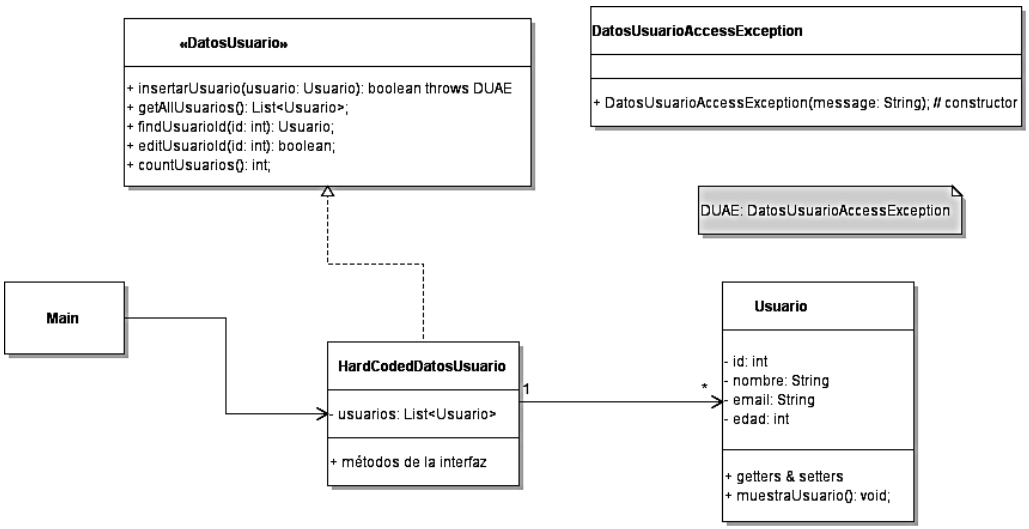
El resultado es 3.

Vemos que la aplicación no ha necesitado entrar en el bloque "catch" porque no se ha producido la posible excepción.

Para ver un ejemplo más práctico, veamos cómo podemos lanzar y capturar una excepción de acceso a datos. Recordemos el ejemplo en el que implementamos la interfaz "DatosUsuario".



En dicho ejemplo en el que nuestro acceso a datos era *hard coded*, imaginemos que la clave primaria de la "supuesta tabla" es el campo "id". Insertemos en la interfaz un nuevo método "insertarUsuario(usuario: Usuario): boolean". Si la clave primaria ya existe en la capa de datos, lanzaremos una nueva excepción "DatosUsuarioAccessException". Veamos cómo queda el nuevo diseño.



La clase del modelo del dominio "Usuario" queda inalterada (ver apartado 3.8). El resto queda de la siguiente forma. Estructura de ficheros.

Podemos ver que hemos creado un paquete "errores", en el que hemos insertado la clase de la excepción. Aparte, hemos modificado la interfaz, su implementación *hard coded*, así como el método *main*. Veamos los cambios.

- Excepción "DatosUsuarioAccessException.java":

```
package errores;
public class DatosUsuarioAccessException extends Exception {
   public DatosUsuarioAccessException() {}
   public DatosUsuarioAccessException(String msg) {
      super(msg);
   }
}
```

- Interfaz "DatosUsuario.java":

```
package dominio;
import errores.DatosUsuarioAccessException;
import java.util.List;

public interface DatosUsuario {
```

```
   public boolean insertarUsuario(Usuario usuario) throws DatosUsuarioAccessException;
   public List<Usuario> getAllUsuarios();
   public Usuario findUsuario(int id);
   public boolean editUsuario(Usuario u);
   public int countUsuarios();
}
```

- "HardCodedDatosUsuario.java":

```
package dominio;
import errores.DatosUsuarioAccessException;
import java.util.ArrayList;
import java.util.List;
public class HardCodedDatosUsuario implements DatosUsuario {
   private List<Usuario> usuarios;
   public HardCodedDatosUsuario() {
      this.usuarios = new ArrayList<Usuario>();
      Usuario u = new Usuario();
      u.setId(1);
      u.setNombre("Ismael");
      u.setEmail("ismael@ismael.com");
      u.setEdad(33);
      this.usuarios.add(u);
      u = new Usuario();
      u.setId(2);
      u.setNombre("Antonio");
      u.setEmail("antonio@antonio.com");
      u.setEdad(44);
      this.usuarios.add(u);
      u = new Usuario();
      u.setId(3);
      u.setNombre("Luis");
      u.setEmail("luis@luis.com");
      u.setEdad(22);
      this.usuarios.add(u);
   }
   @Override
   public boolean insertarUsuario(Usuario usuario) throws
         DatosUsuarioAccessException {
      int id = usuario.getId();
      boolean encontrado = false;
      for(Usuario u: this.usuarios) {
         if (u.getId() == id) {
            encontrado = true;
            break;
         }
```

```
        }
        if(encontrado) {
            throw new DatosUsuarioAccessException("El id "
                + "ya se encuentra en la base de datos");
        } else {
            Usuario nuevoUsuario = new Usuario();
            nuevoUsuario.setId(usuario.getId());
            nuevoUsuario.setNombre(usuario.getNombre());
            nuevoUsuario.setEmail(usuario.getEmail());
            nuevoUsuario.setEdad(usuario.getEdad());
            this.usuarios.add(nuevoUsuario);
        }
        return true;
    }
    @Override
    public List<Usuario> getAllUsuarios() {
        return this.usuarios;
    }
    @Override
    public Usuario findUsuario(int id) {
        for (Usuario u: this.usuarios) {
            if (id == u.getId()) {
                return u;
            }
        }
        return null;
    }
    @Override
    public boolean editUsuario(Usuario u) {
        for (Usuario us: this.usuarios) {
            if (u.getId() == us.getId()) {
                us.setNombre(u.getNombre());
                us.setEmail(u.getEmail());
                us.setEdad(u.getEdad());
                return true;
            }
        }
        return false;
    }
    @Override
    public int countUsuarios() {
        return this.usuarios.size();
    }
}
```

- Clase: "Main.java":

```
package principal;
import dominio.DatosUsuario;
import dominio.HardCodedDatosUsuario;
import dominio.Usuario;
import errores.DatosUsuarioAccessException;
public class Main {
  public static void main(String[] args) {
      DatosUsuario du = new HardCodedDatosUsuario();
      // El id 2 ya existe.
      Usuario u = new Usuario();
      u.setId(2);
      u.setNombre("Nicolás");
      u.setEmail("nicolas@nicolas.com");
      u.setEdad(20);
    try {
      boolean correcto = du.insertarUsuario(u);
      if (correcto) System.out.println("El usuario se ha insertado bien".);
    } catch (DatosUsuarioAccessException ex) {
      System.out.println("Error: "+ex.getMessage()+"".);
    }
  }
}
```

Al existir un usuario con id = 2 en la base de datos, se lanza la excepción y se captura en el método *main*.

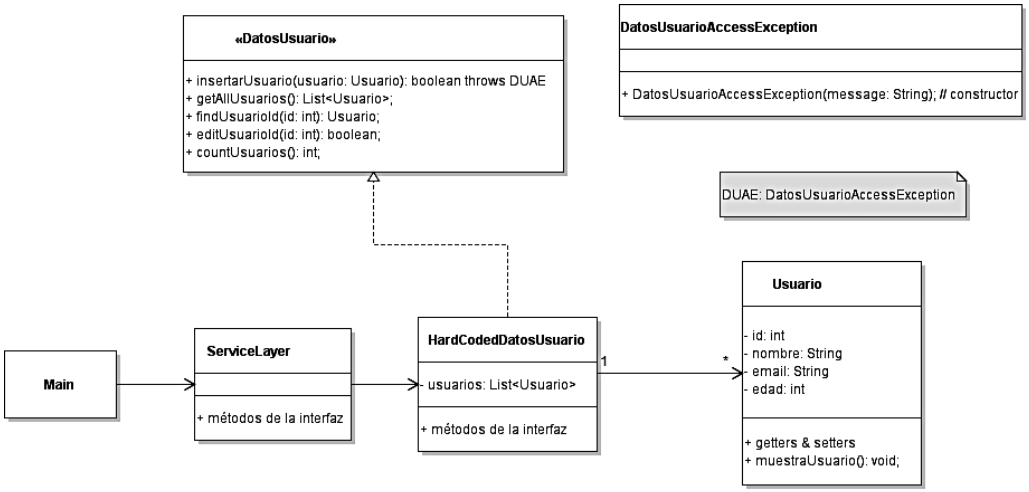
Veamos la salida por consola:

Error: El id ya se encuentra en la base de datos.

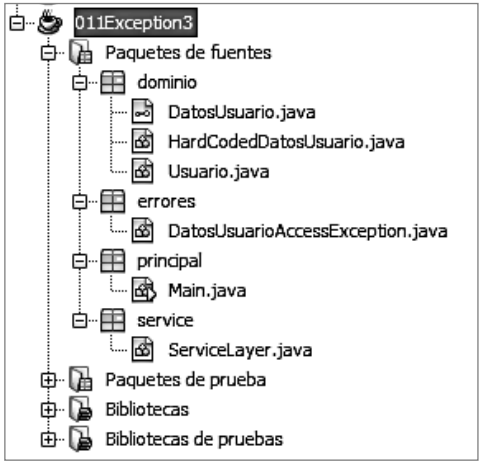
Sin embargo, para el id = 4 (que no se ha usado), la salida del programa será la siguiente:

El usuario se ha insertado bien.

Para finalizar con las excepciones, veamos cómo podemos tener métodos que capturen excepciones y no tengan una cláusula "try... catch", sino que directamente lancen las excepciones al método que las ha llamado. Esta técnica es común para la capa de servicio. Esto lo conseguimos con la cláusula "throws", sin necesidad de que la excepción haya sido creada en el método. Para ver un caso práctico, imaginemos una capa de servicio tonta, que lo único que haga sea llamar a la capa de datos *hard coded* que hemos desarrollado.



En la clase "ServiceLayer" el método "insertarUsuario" lanza la excepción sin crearla. Veamos el código reestructurado.



Las dos clases que nos interesan son las siguientes:

- Clase "ServiceLayer.java" (es una capa de servicio tonta, no hace nada):

```
package service;
import dominio.DatosUsuario;
import dominio.HardCodedDatosUsuario;
import dominio.Usuario;
import errores.DatosUsuarioAccessException;
import java.util.List;
public class ServiceLayer {
   private DatosUsuario datosUsuario;
```

```
    public ServiceLayer() {
        this.datosUsuario = new HardCodedDatosUsuario();
    }
    public boolean insertarUsuario(Usuario usuario)
                        throws DatosUsuarioAccessException {
        return this.datosUsuario.insertarUsuario(usuario);
    }
    public List<Usuario> getAllUsuarios() {
        return this.datosUsuario.getAllUsuarios();
    }
    public Usuario findUsuario(int id) {
        return this.datosUsuario.findUsuario(id);
    }
    public boolean editUsuario(Usuario u) {
        return this.datosUsuario.editUsuario(u);
    }
    public int countUsuarios() {
        return this.datosUsuario.countUsuarios();
    }
}
```

- Clase "Main.java":

```
package principal;
import dominio.Usuario;
import errores.DatosUsuarioAccessException;
import java.util.List;
import service.ServiceLayer;
public class Main {
    public static void main(String[] args) {
            ServiceLayer sl = new ServiceLayer();
            Usuario u = new Usuario();
            u.setId(4);
            u.setNombre("Nicolás");
            u.setEmail("nicolas@nicolas.com");
            u.setEdad(20);
        try {
            boolean correcto = sl.insertarUsuario(u);
            if (correcto) System.out.println("El usuario se ha "
                    + "insertado bien".);
        } catch (DatosUsuarioAccessException ex) {
            System.out.println("Error: "+ex.getMessage()+"".);
        }
        List<Usuario> usuarios = sl.getAllUsuarios();
        for (Usuario us: usuarios) {
            System.out.println("***********");
```

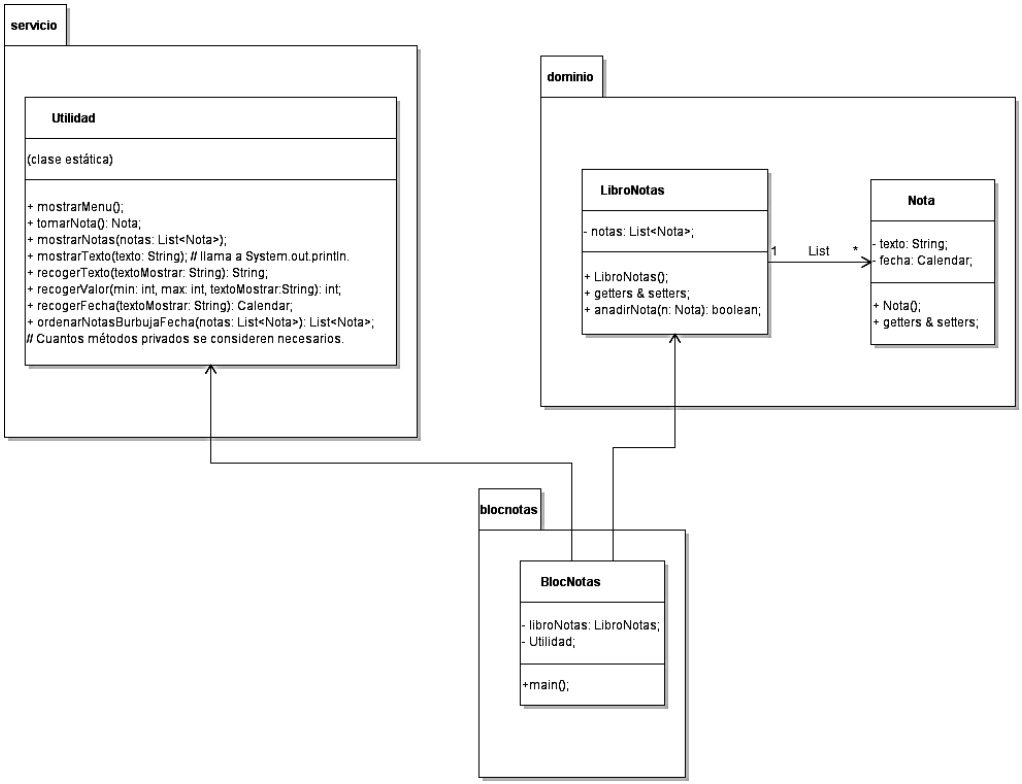
```
            us.muestraUsuario();
        }
    }
}
```

La salida por consola de la aplicación es la siguiente:

El usuario se ha insertado bien.
****
Id: 1.
Nombre: Ismael.
Email: ismael@ismael.com.
Edad: 33.
****
Id: 2.
Nombre: Antonio.
Email: antonio@antonio.com.
Edad: 44.
****
Id: 3.
Nombre: Luis.
Email: luis@luis.com.
Edad: 22.
****
Id: 4.
Nombre: Nicolás.
Email: nicolas@nicolas.com.
Edad: 20.

3.10 Ejercicio 1

Se propone al lector implementar una aplicación de consola a modo de bloc de notas. La aplicación constará de un menú en el que el usuario podrá elegir si introducir una nota nueva o ver las notas existentes. Cuando introducimos una nota nueva, primero introducimos la fecha (ya que el bloc funcionará a modo de calendario, con notas ordenadas por fecha) y luego introducimos el texto. Si elegimos ver las notas, veremos el listado de notas por orden cronológico. El esquema de clases y de paquetes que se propone es el siguiente:



*Nota: En la clase estática usamos el método de ordenación burbuja.

Instalación del entorno servidor de aplicaciones Oracle WebLogic Server

CAPÍTULO 4

Una vez estudiadas las nociones básicas de Java (plataforma JSE), vamos a comenzar a instalar nuestro entorno de trabajo para la JEE. No nos confundamos. La versión de NetBeans que hemos instalado en nuestra máquina nos da acceso al conjunto de librerías y clases que componen la JEE. Pero nuestras aplicaciones empresariales (cuya principal característica es la conectividad) deben correr (ejecutarse) en un servidor de aplicaciones empresariales. Existen varios servidores de aplicaciones para la JEE. Entre los de libre distribución nos encontramos: JOnAS, JBoss (últimas versiones conocidas como WildFly), Geronimo, TomEE o GlassFish (de Oracle, instalada con nuestro NetBeans), entre otras. Entre los servidores de aplicaciones propietarios están WebSphere (de IBM) y WebLogic (de Oracle). El servidor de aplicaciones WebLogic, aun siendo propietario, puede ser descargado bajo la licencia OTN *(Oracle Technology Network License Agreement)*. Es una versión completa pero en modo prueba, sólo disponible para versiones de desarrollo. En el sitio web de Oracle, bajo las restricciones de Oracle WebLogic, podemos leer lo siguiente (versión original en inglés):

"Oracle grants You a nonexclusive, nontransferable, limited license to internally use the Programs, subject to the restrictions stated in this Agreement, only for the purpose of developing, testing, prototyping, and demonstrating Your application and only as long as Your application has not been used for any data processing, business, commercial, or production purposes, and not for any other purpose".

La traducción al castellano del anterior texto sería la siguiente:

"Oracle le concede una licencia ni exclusiva ni intransferible y limitada para utilizar internamente los programas, con sujeción a las restricciones establecidas en este acuerdo, sólo con el propósito de desarrollo, prueba, prototipo y demostración de su aplicación, y sólo durante el tiempo en que su aplicación no sea utilizada para ningún propósito de procesamiento de datos, negocios, comercial o de producción; y para ningún otro propósito".

Digamos que cumplimos la restricción del acuerdo porque nuestro uso va a ser sólo para desarrollo, aprendizaje y enseñanza de nuestro propio código. No vamos a usar Oracle WebLogic Server para fines de producción de aplicaciones reales ni comerciales (no vamos a vender *software* que se ejecute sobre Oracle WebLogic).

¿Por qué usamos WebLogic? La respuesta es simple: por su robustez. Es un servidor de aplicaciones desarrollado desde el año 1997 por la compañía BEA Systems, adquirida posteriormente por Oracle (es el servidor de aplicaciones propietario de Oracle). Al ser un servidor propietario y empaquetado, va a ser muy sencillo tanto su uso como el despliegue de nuestras aplicaciones empresariales. El lector, una vez conozca las características de este servidor de aplicaciones, y sobre todo de la JEE, podrá migrar sus aplicaciones o desarrollarlas para que se ejecuten en el servidor de aplicaciones que estime conveniente.

Pero ¿qué es realmente un servidor de aplicaciones?

La tecnología de servidores de aplicaciones, sobre la que se basa el desarrollo de aplicaciones empresariales de la JEE, nos proporciona un conjunto de funcionalidades básicas:

- Centralización de la lógica de negocio, usando la API de Enterprise JavaBeans (EJB). Los EJB harán uso de la *Java Persistence API* (JPA) o API de Persistencia de Java para persistir los objetos. Gracias a la tecnología de EJB, tendremos acceso a otro subconjunto de tecnologías que nos permitirán:
  - Tener facilidad para la conectividad basándonos en el desarrollo de componentes *software* que se intercomunican.
  - Acceso universal al árbol de objetos, conexiones y componentes desplegado a modo de árbol de directorios sobre el servidor. A dicho árbol lo denominamos árbol JNDI *(Java Naming and Directory Interface)*.
  - Olvidarnos en gran medida de los problemas arquitectónicos de la aplicación.
- Contenedor de *servlets*.
- Un largo etcétera, del cual no pretendemos ser exhaustivos.

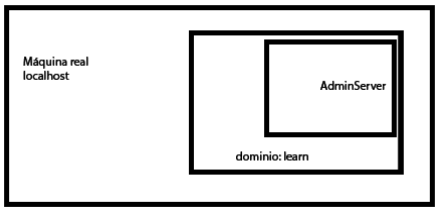
Denominamos *lógica de negocio* al conjunto de datos y operaciones sobre esos datos aplicables al dominio de un cliente concreto. Por ejemplo, para un cliente que necesite una aplicación de facturación, tendríamos un conjunto de datos: productos, servicios, totales..., y un conjunto de operaciones sobre estos datos: calcular precio, calcular regla de impuestos, calcular total, crear factura...

La JPA es una especificación JSR que permite crear una base de datos relacional (y gestionarla) a partir de un diagrama de clases diseñado siguiendo los principios de la programación orientada a objetos. Esto se consigue gracias al patrón de mapeo objeto-relacional (también denominado *ORM*: *Object-Relational mapping*).

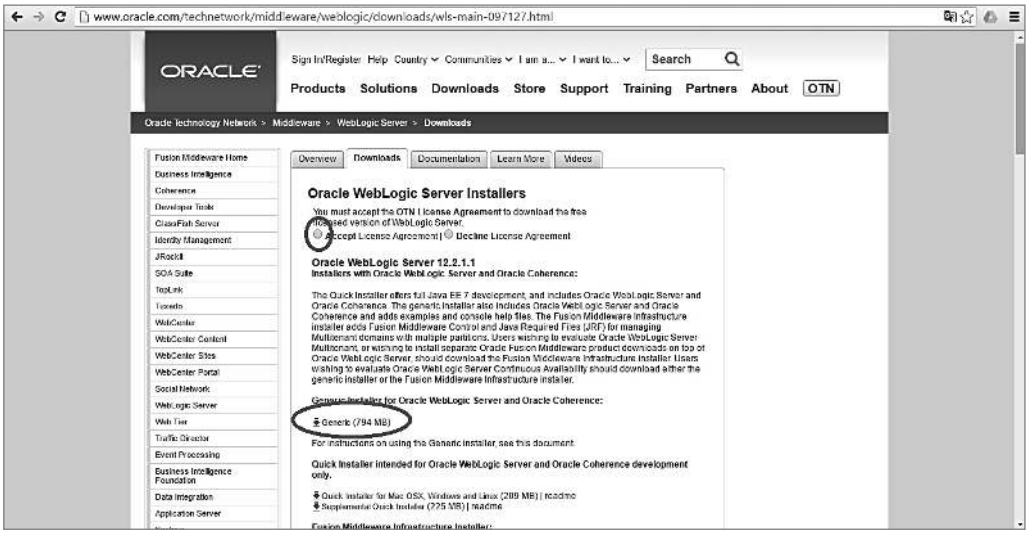
Facilidad para la conectividad: el servidor de aplicaciones nos permite la conectividad universal mediante un conjunto de tecnologías: servicios web (SOAP o RESTful), RMI, CORBA, JMS...

Acceso a los objetos, conexiones y artefactos desplegados sobre el servidor de aplicaciones mediante una búsqueda sobre el contexto de la aplicación y sobre el árbol JNDI, estando en la misma máquina virtual o en una máquina virtual diferente.

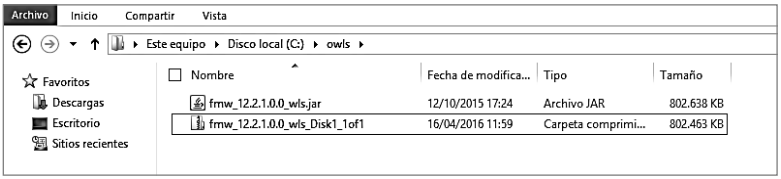
El servidor de aplicaciones debe contener un dominio para el espacio de nombres. En nuestro caso tendremos la configuración más simple:



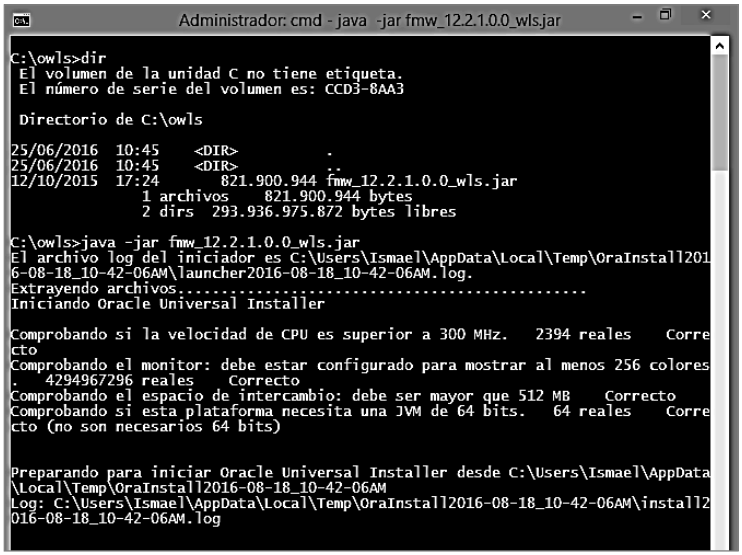
Para instalar Oracle WebLogic Server nos vamos a la siguiente URL: "http://www.oracle.com/technetwork/middleware/weblogic/downloads/wls-main-097127.html". Debemos aceptar la licencia OTN y descargarnos la versión genérica de 794 MB que contiene el instalador gráfico. La versión actual de WebLogic Server es la 12c.



Una vez descargada, procedemos con la instalación. Se nos habrá descargado un archivo comprimido ".zip". En nuestro caso "fmw_12.2.1.0.0_wls_Disk1_1of1.zip". Lo descomprimimos (obtendremos un fichero ".jar" ejecutable) y podremos eliminar el archivo ".zip" descargado.



Acto seguido vamos a la consola de comandos y ejecutamos el fichero ".jar".



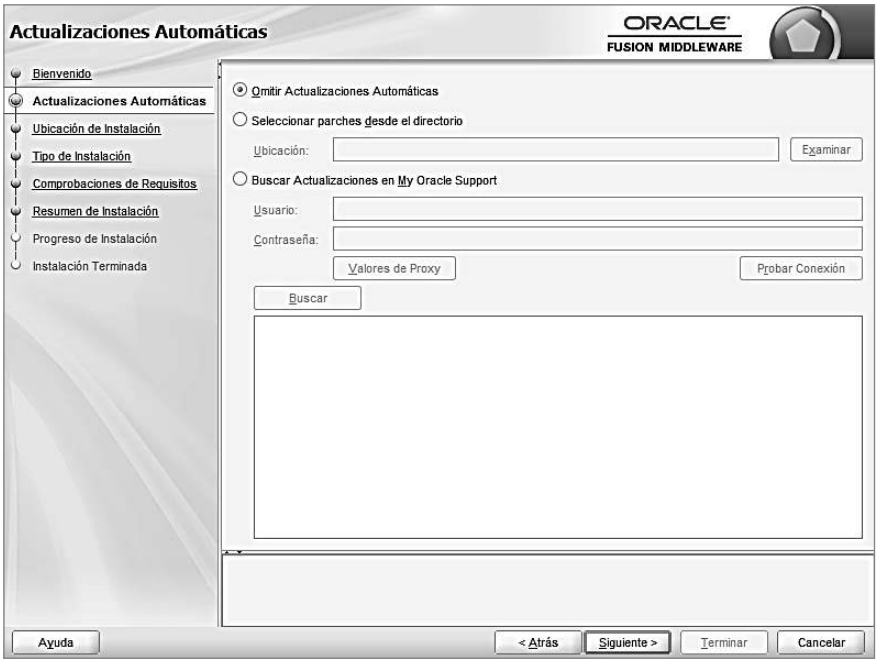
El instalador que hemos ejecutado desde la línea de comandos lanzará el instalador gráfico.

Vemos a continuación la sucesión de pantallas del instalador del servidor de aplicaciones.

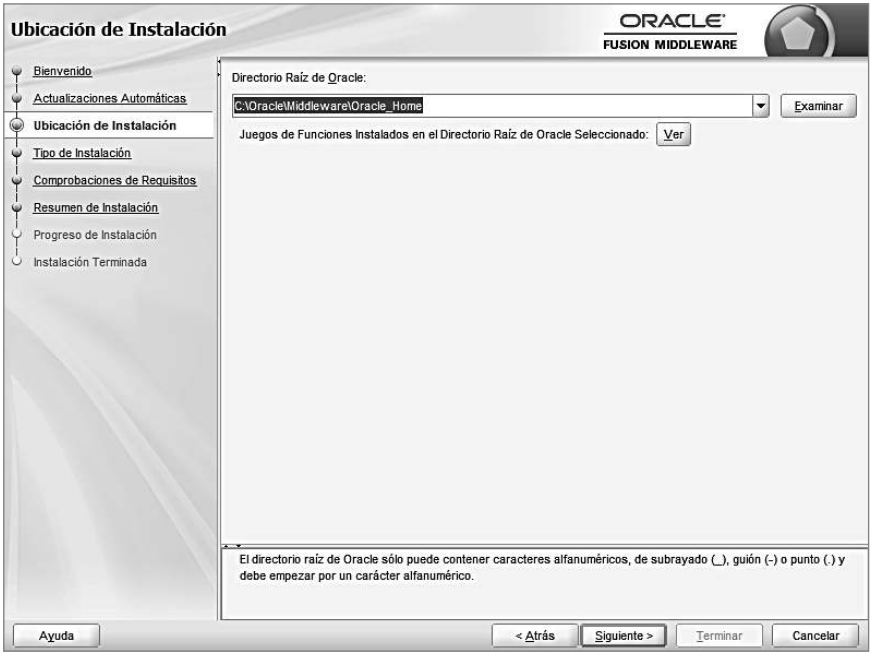
Pulsamos en siguiente, el conjunto de pasos que vamos a seguir serán los que se muestran en la columna de la izquierda.

- Actualizaciones automáticas.
- Ubicación de la instalación.
- Tipo de instalación.
- Comprobaciones de requisitos.
- Resumen de instalación.
- Progreso de instalación.
- Fin de la instalación: instalación terminada.

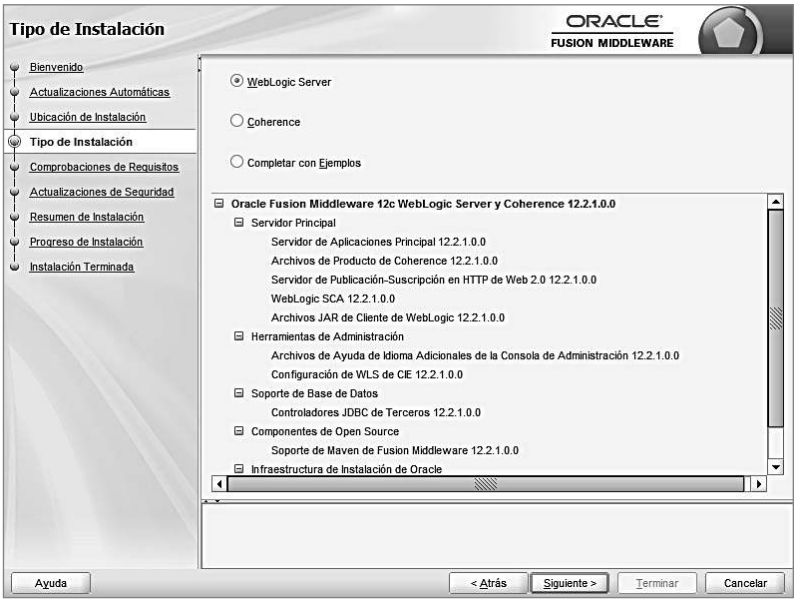
Seleccionamos "omitir actualizaciones automáticas".



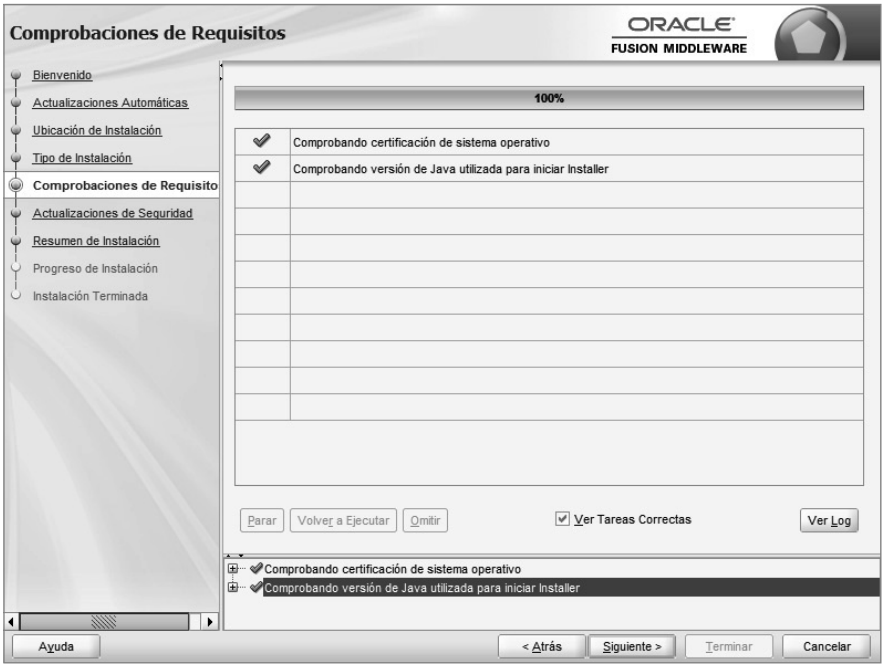
Indicamos el directorio en el que queremos instalar el servidor de aplicaciones.



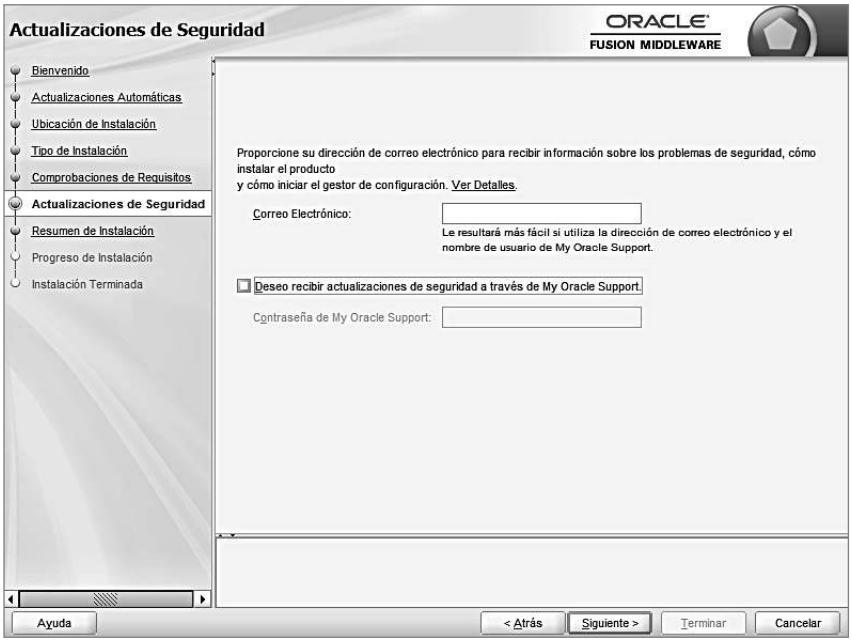
Seleccionamos el tipo de instalación: WebLogic Server. Coherence es una herramienta de Oracle que permite fácilmente la escalabilidad de aplicaciones.



Comprobación de requisitos del sistema.

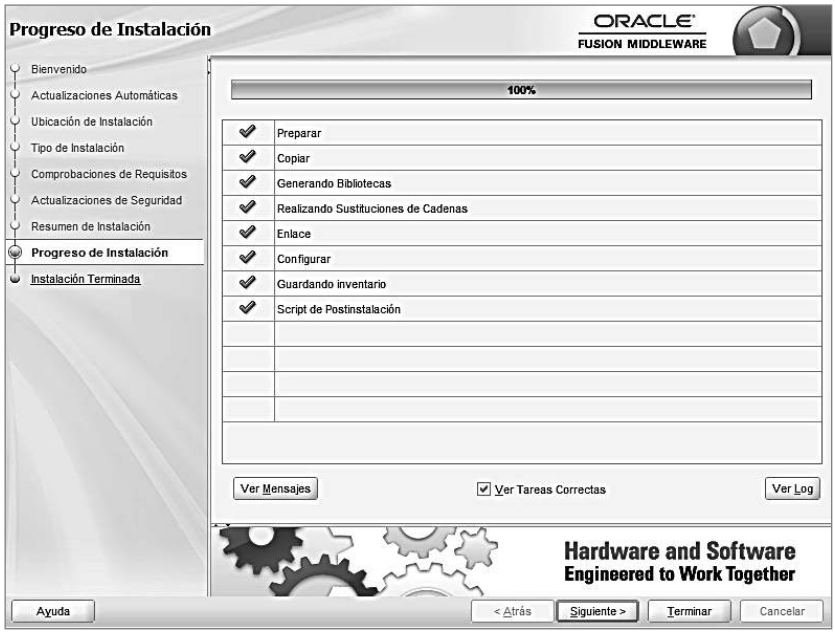


Deseleccionamos la opción de recibir actualizaciones de seguridad en nuestra cuenta de Oracle.

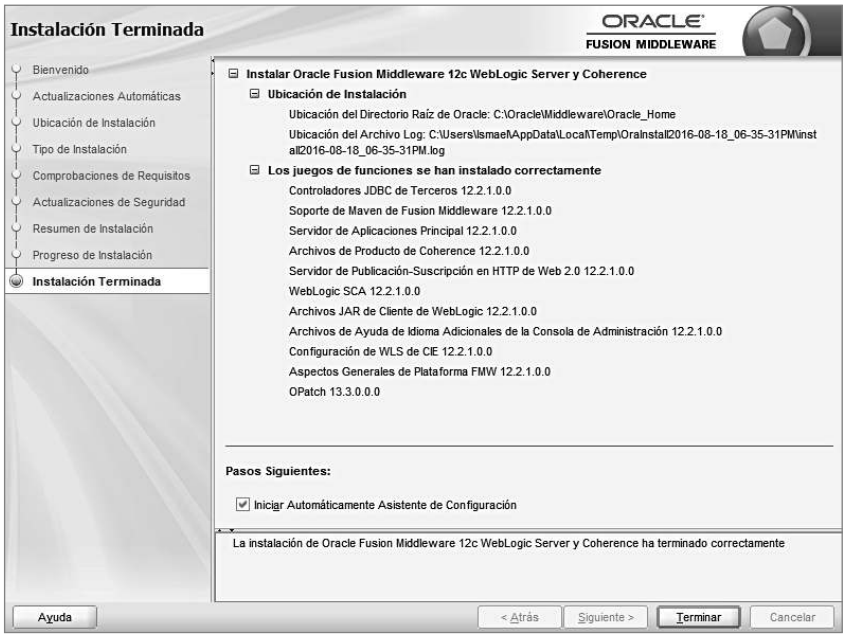


Continuamos hacia la pantalla de resumen de la instalación.

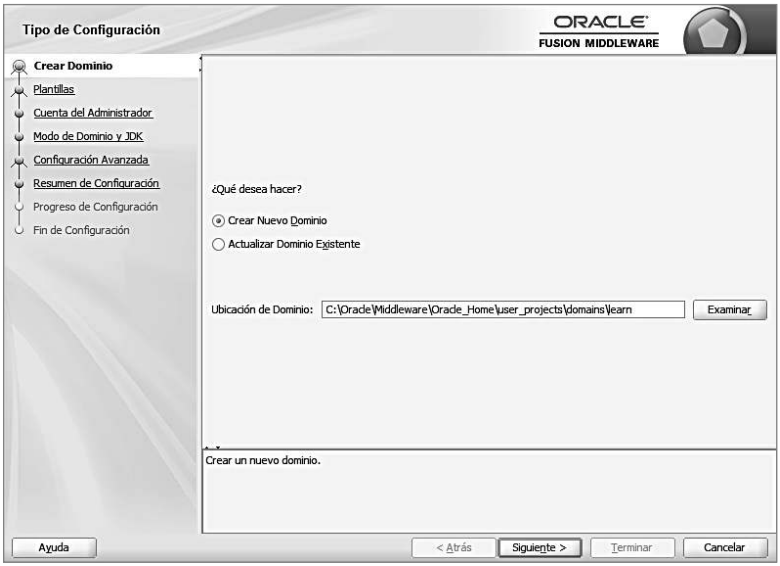
El servidor de aplicaciones debe terminar de instalarse exitosamente.



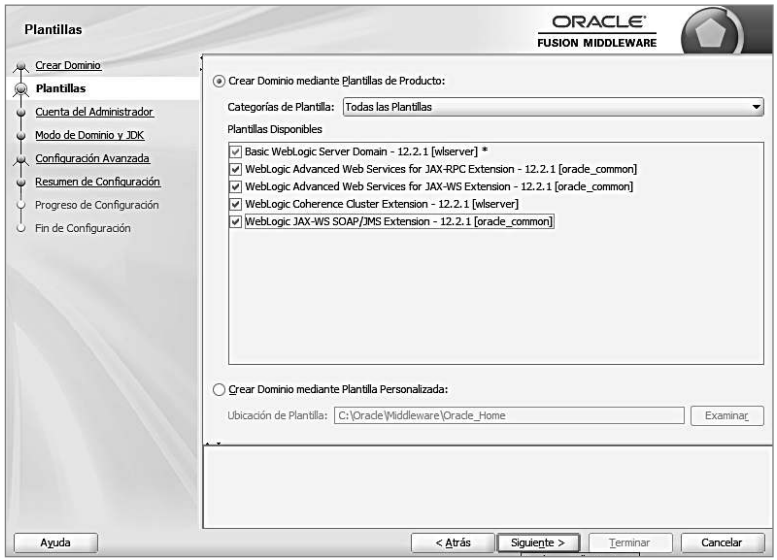
Pulsamos en "Siguiente". Observamos que aparece la opción "Pasos siguientes" al final de la pantalla que debemos seleccionar para configurar nuestro dominio de trabajo.



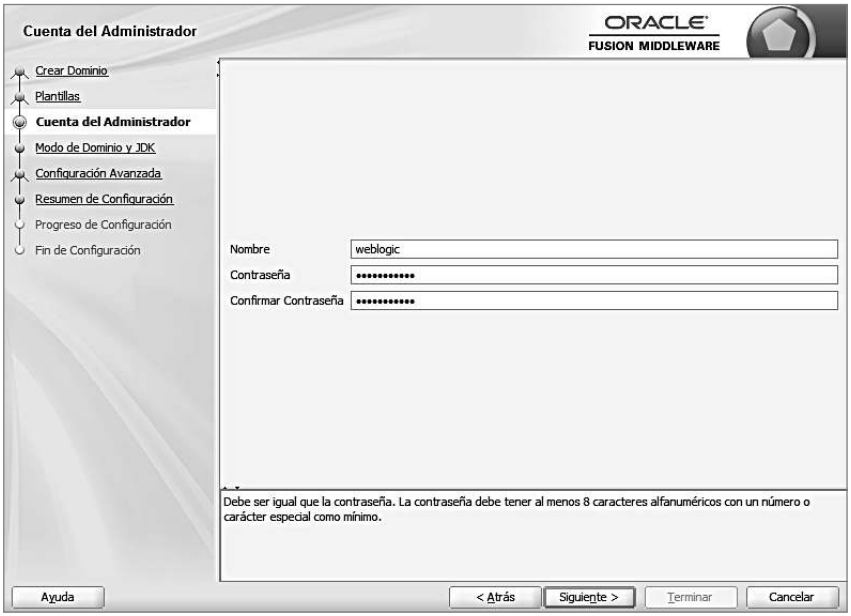
Un dominio de trabajo es aquel bajo el que vamos a configurar nuestro servidor de aplicaciones. Una máquina real puede tener varios dominios, y cada dominio, a su vez, varios servidores virtuales. Para trabajar en nuestra máquina real necesitamos al menos un dominio y su servidor virtual por defecto "AdminServer". Vamos a crear un dominio que se llame *learn* ("aprender"). Al final de la ruta indicamos el nombre del dominio que queremos crear.



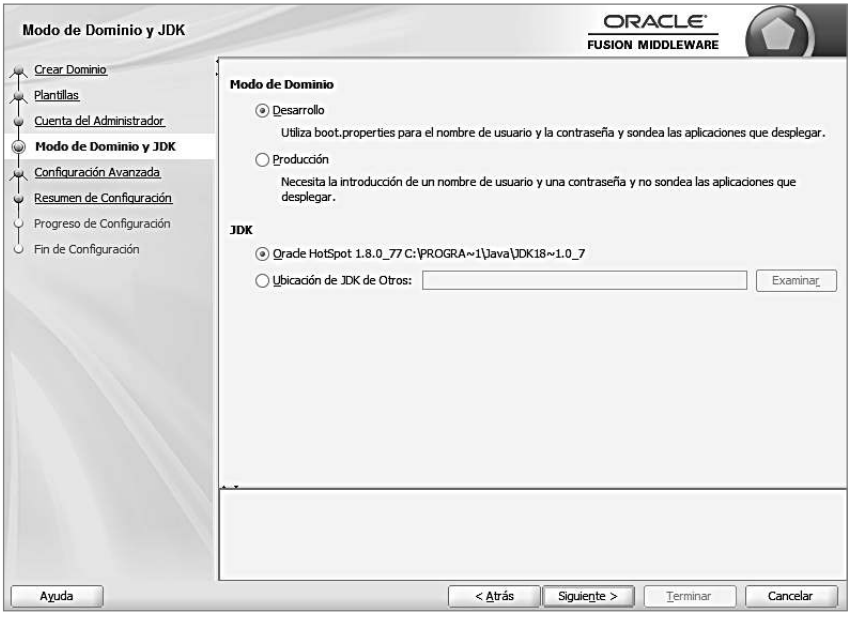
A continuación le indicamos que queremos seleccionar todas las plantillas para conectividad del dominio.



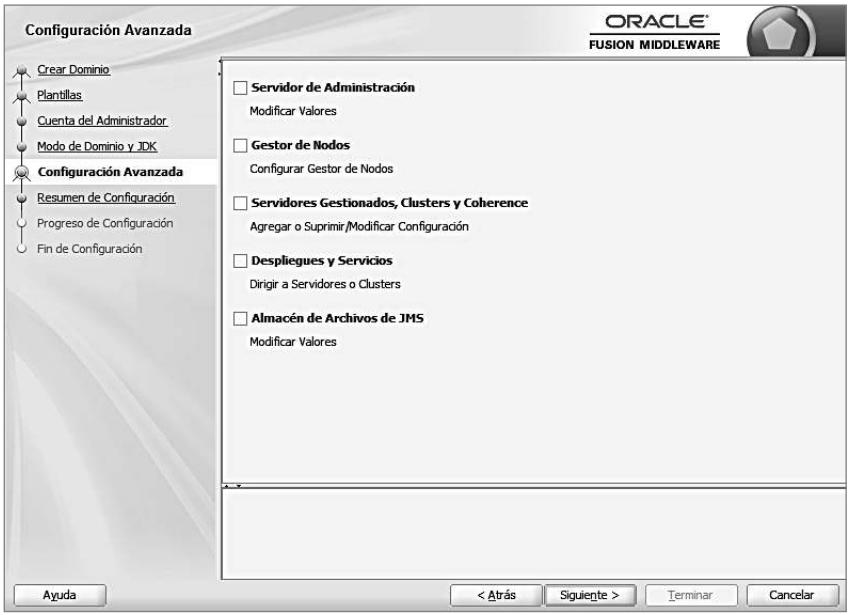
A continuación indicamos el usuario y la contraseña de administración del servidor de aplicaciones. Será necesario para acceder al panel de gestión y para conectarnos a él desde NetBeans.



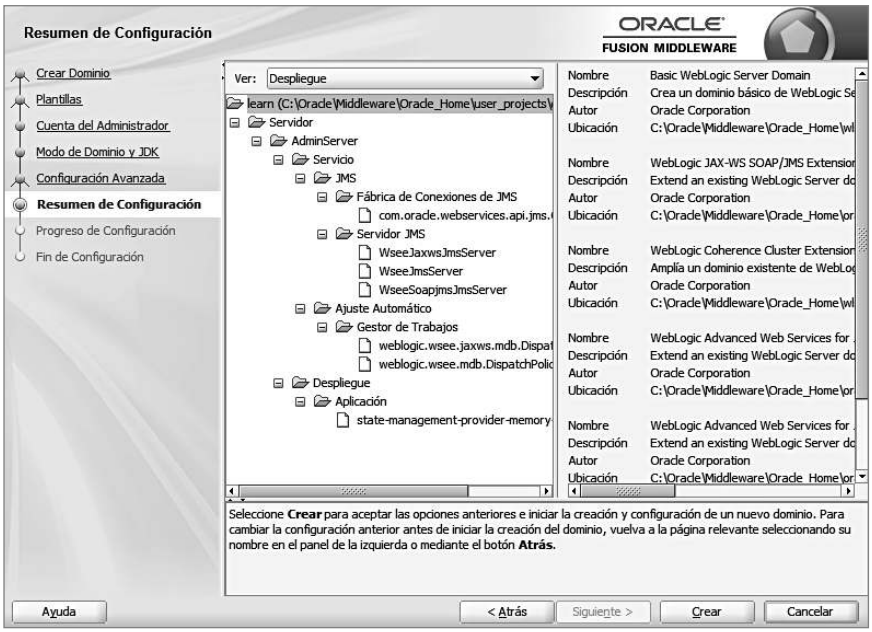
Instalamos el dominio en modo desarrollo y le indicamos al configurador dónde se encuentra nuestra JDK.



En la pantalla "Configuración Avanzada" dejamos todas las opciones por seleccionar.



El configurador nos muestra un mensaje con la configuración del dominio. Es una pantalla de información, simplemente pulsamos en "Crear" para crear el dominio con la configuración que hemos ido indicando.



Una vez terminado el proceso, nos aparece la siguiente pantalla.

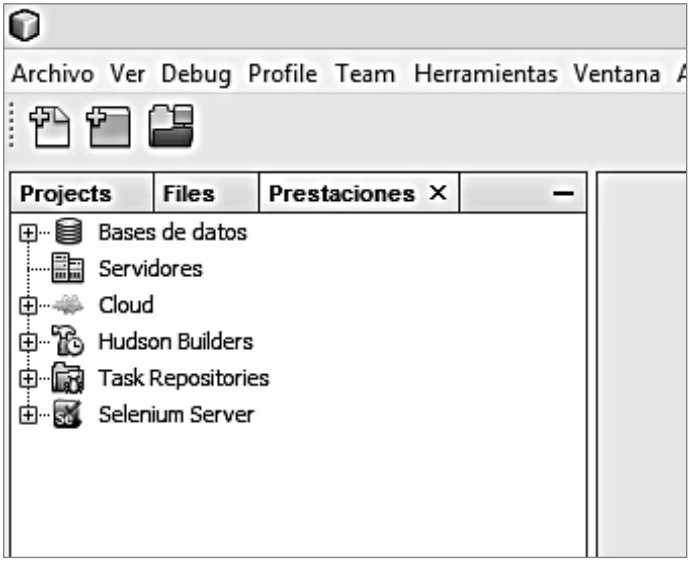
Pulsando en "Siguiente", vemos la configuración del servidor.

Bien. Hemos instalado nuestro servidor de aplicaciones Oracle WebLogic Server. Veamos ahora cómo podemos desarrollar aplicaciones en nuestro IDE (NetBeans) y desplegarlas en nuestro servidor de aplicaciones. El primer paso que debemos dar es abrir NetBeans.

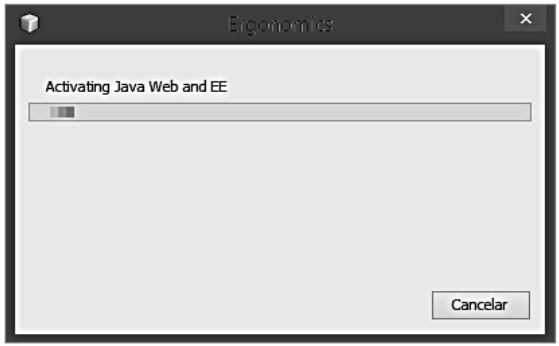
En la columna de la izquierda (que, una vez abierto NetBeans, se encuentra vacía), tenemos tres pestañas:

- Proyectos.
- Archivos.
- Prestaciones.

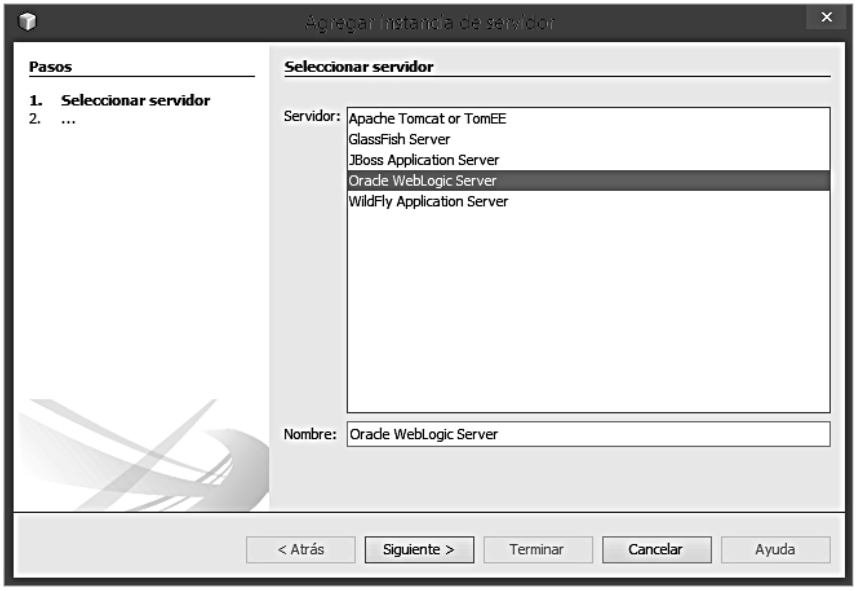
Vayamos a la pestaña de prestaciones y cliquemos en la opción "Servidores".



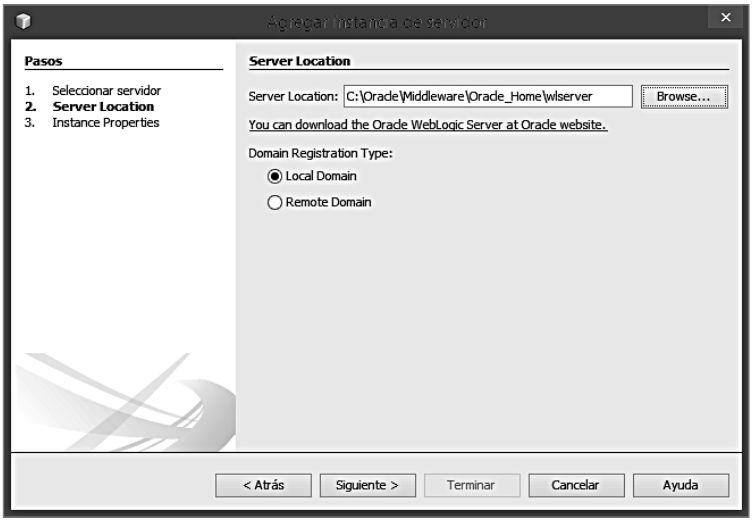
Una vez que hemos hecho lo comentado, nos aparecerá el siguiente mensaje indicando que NetBeans está activando las características de Java EE (este mensaje no aparecería si NetBeans ya lo hubiera hecho).



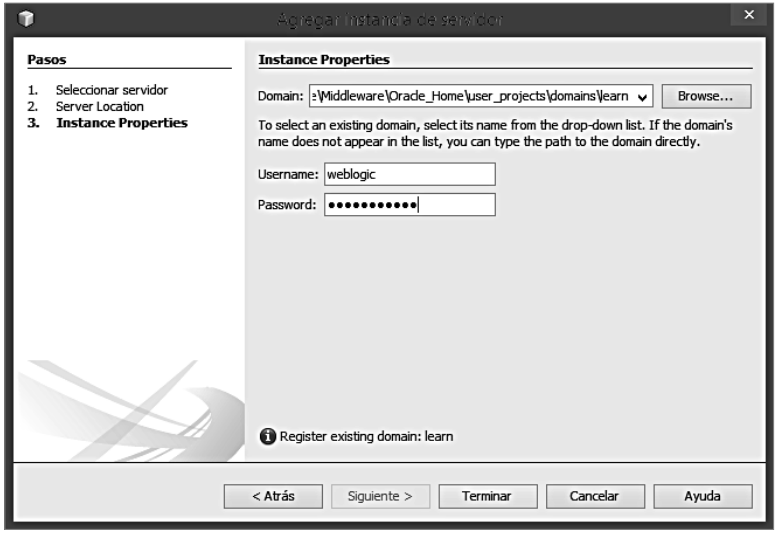
A continuación se nos pregunta por el tipo de servidor que queremos enlazar desde NetBeans. En nuestro caso y como bien sabemos, es Oracle WebLogic Server.



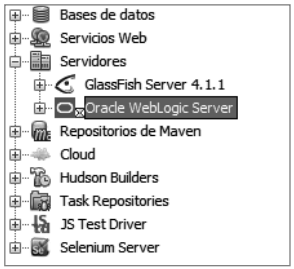
Nos pide la ruta de nuestro disco duro en la que tenemos instalado el servidor de aplicaciones. Se la indicamos. En "Tipo de Registro de Dominio", indicamos "Local Domain".



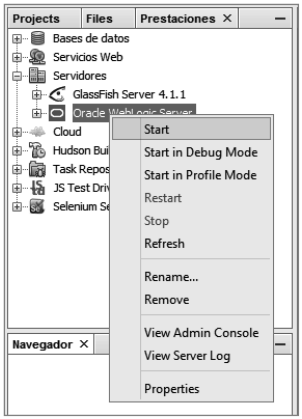
Por defecto, ha detectado el dominio "learn" en nuestra instalación. A continuación le indicamos el usuario y la contraseña de administración del servidor, aquella que indicamos cuando creamos el dominio.



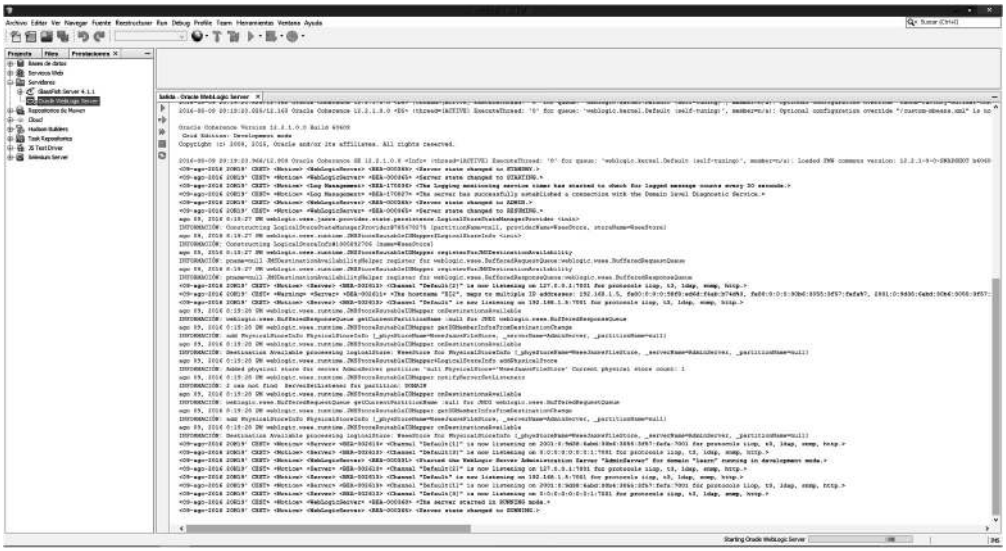
Para finalizar la instalación del servidor con su dominio en NetBeans, pulsamos en "Terminar". Si ahora acudimos a la pestaña de "Prestaciones" en la primera columna, encontraremos el servidor "Oracle WebLogic Server" instalado.



Para iniciar el servidor nos posicionamos encima de él y seleccionamos "Start".



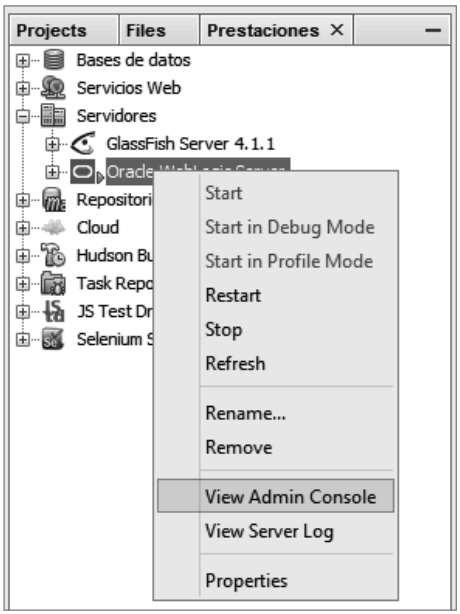
Podremos ver en la salida del sistema el *log* de operaciones de inicio del servidor.



La operación de arranque estará terminada una vez que veamos el siguiente mensaje:

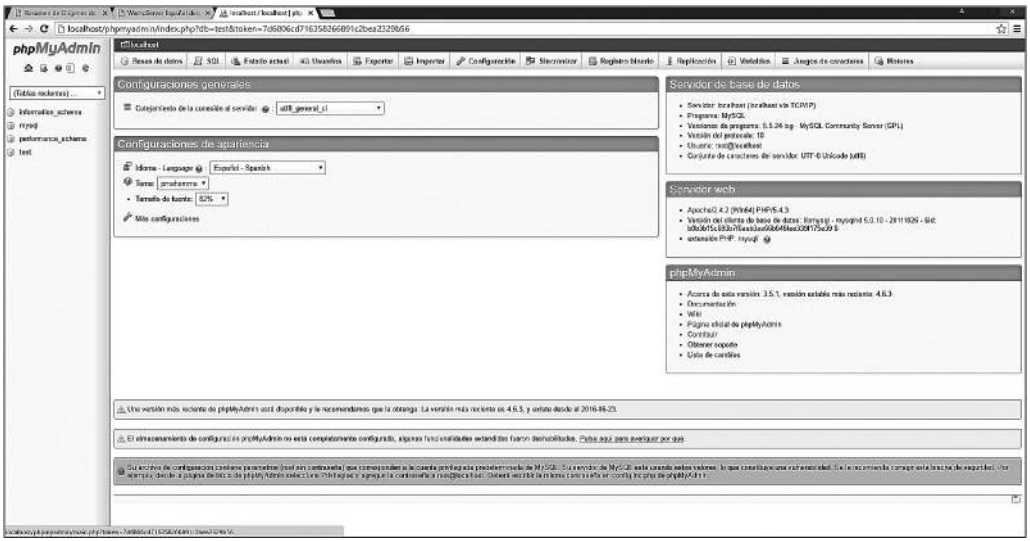
"<Notice> <WebLogicServer> <BEA-000365> <Server state changed to RUNNING.>".

Para poder gestionar el servidor, debemos seleccionar la opción "Ver la Consola de Administración".



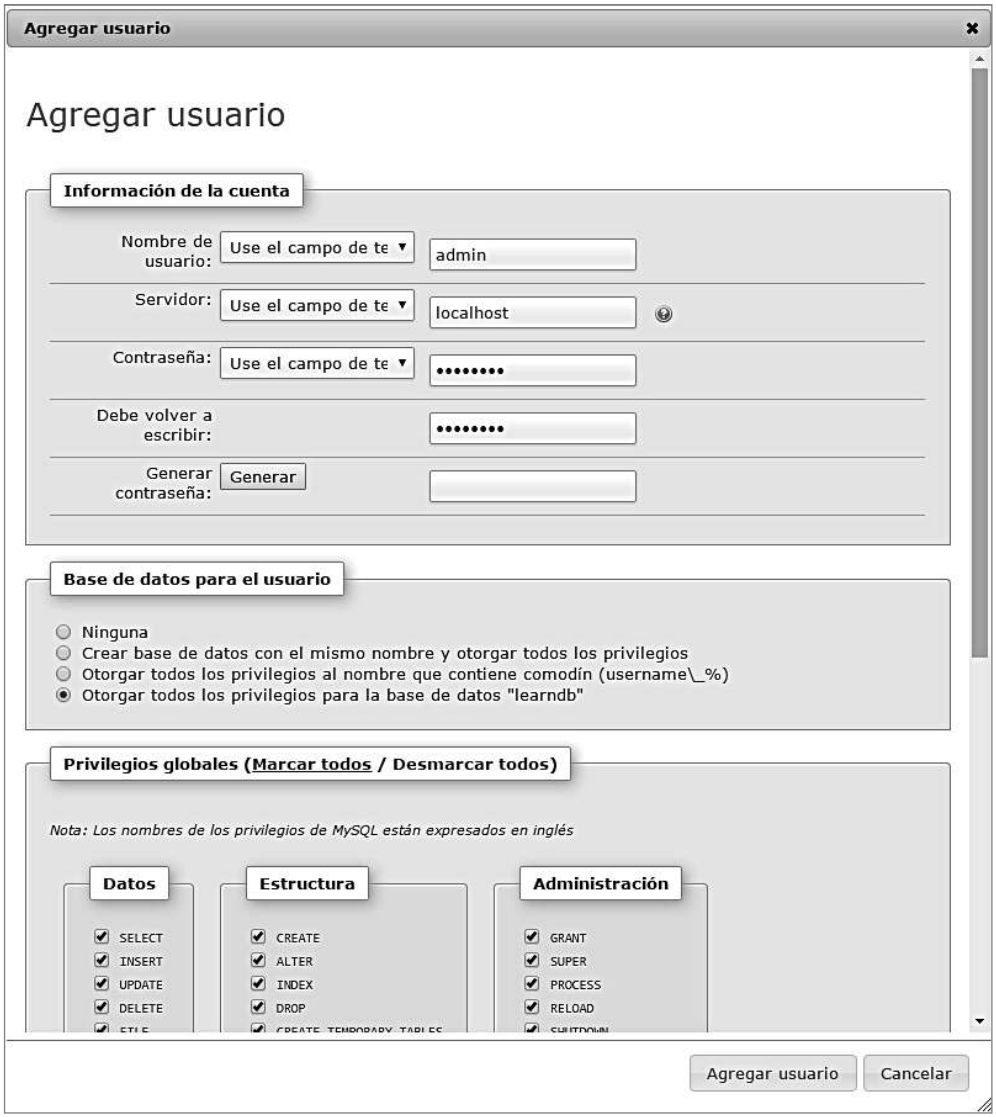
En dicho caso accedemos a la ventana de administración del servidor de aplicaciones. Como se ha comentado, nos encontramos en el dominio "learn". Nuestras aplicaciones, conexiones y demás elementos de JEE se desplegarán sobre el servidor virtual "AdminServer" de dicho dominio.

Para poder desarrollar aplicaciones empresariales, necesitamos conectarnos a una base de datos. Vamos a crear una nueva base de datos llamada *learndb* y vamos a enlazarla a nuestro servidor de aplicaciones para cuando nuestras aplicaciones hagan uso de ella. Supongamos un gestor de base de datos SQL, en este caso MySQL. Instalamos a nivel local *(localhost)* tanto el SGBD como un cliente, en este caso el cliente web "phpMyAdmin".

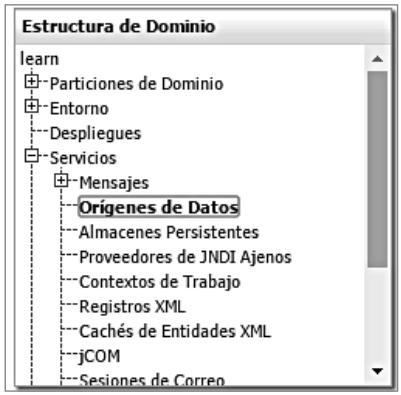


Acudiendo al apartado de "Bases de datos", creamos una nueva base de datos "learndb".

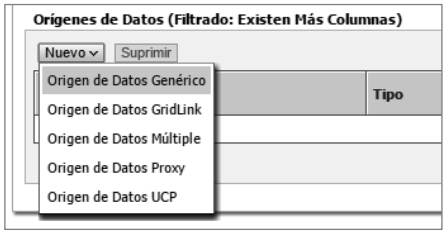
Creamos el usuario para la base de datos y le otorgamos todos los privilegios posibles.



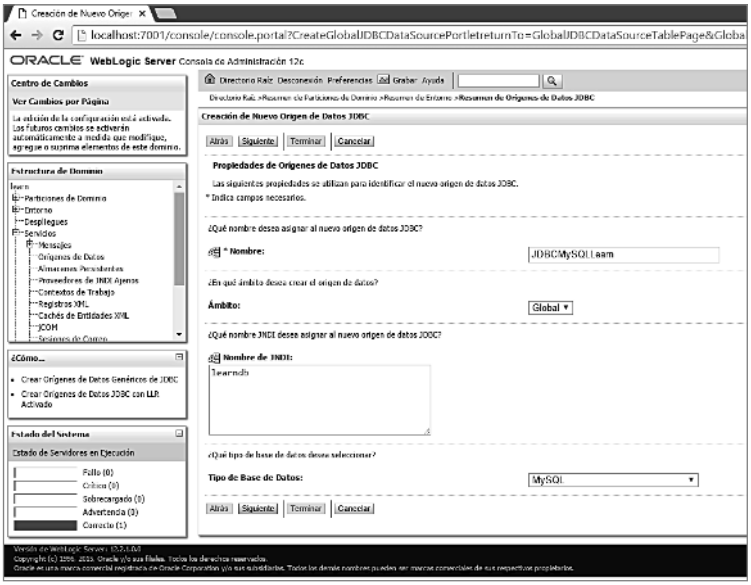
Acto seguido volvemos al servidor de aplicaciones y en la ventana "Estructura de Dominio", seleccionamos la opción "Servicios" y a su vez la subopción "Orígenes de Datos".



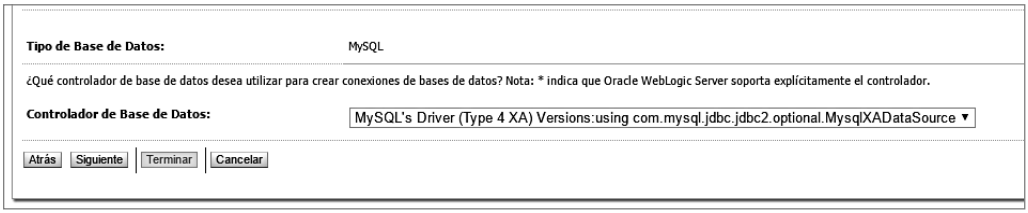
Para enlazar con la base de datos que tenemos pensada debemos escoger "Origen de Datos Genérico".



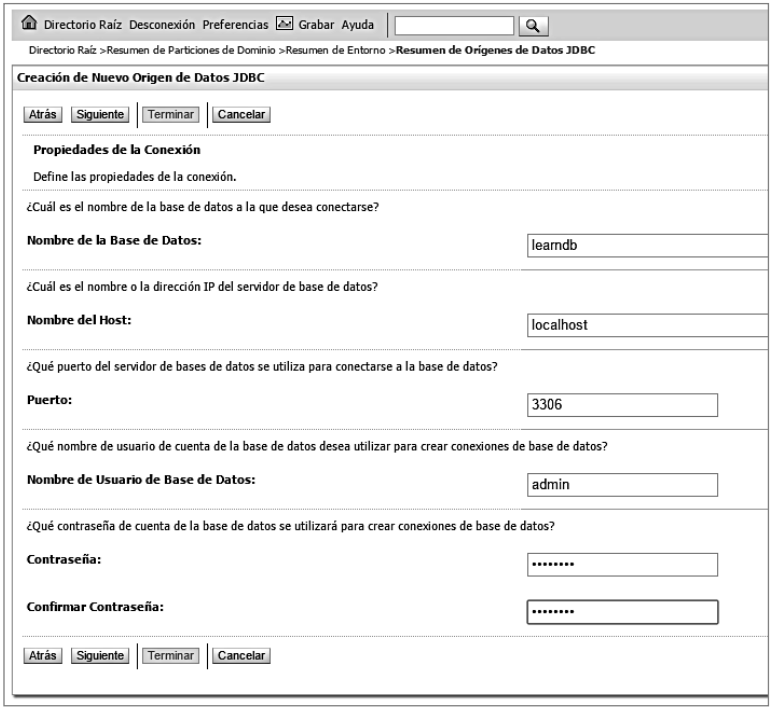
En la siguiente ventana debemos coger el nombre ("JDBCMySQLLearn"), nombre JNDI (la hemos llamado igual, "learndb") y el tipo de SGBD (en nuestro caso "MySQL").



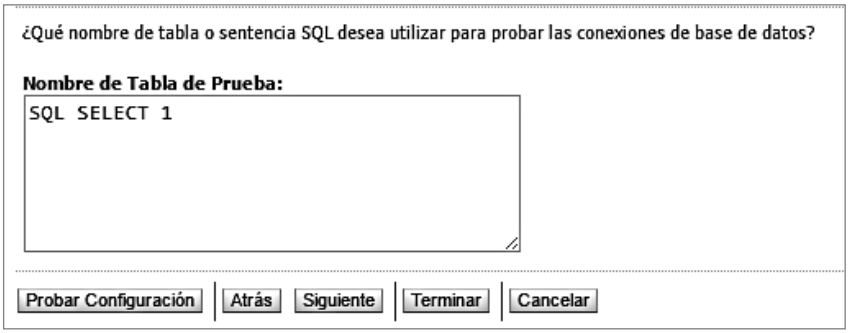
En la siguiente ventana debemos escoger el *driver* o controlador que por defecto nos ofrece el servidor para MySQL.



Le indicamos las opciones de conexión a la base de datos.

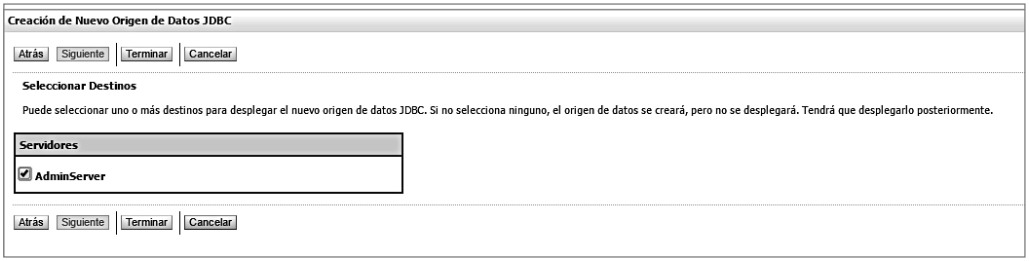


Realizamos una conexión de prueba.

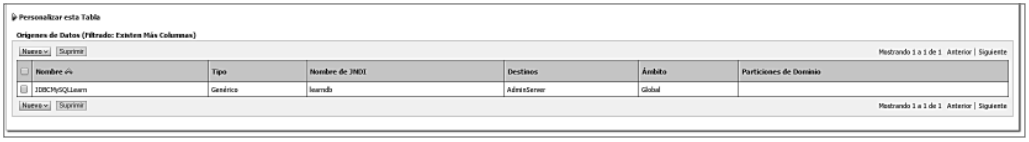


Es importante que en este paso tengamos una conexión correcta así como que presionemos el botón "Siguiente", ya que debemos desplegar en la siguiente ventana.

Desplegamos sobre "AdminServer".



Finalmente, veremos nuestro origen de datos desplegado sobre "AdminServer".



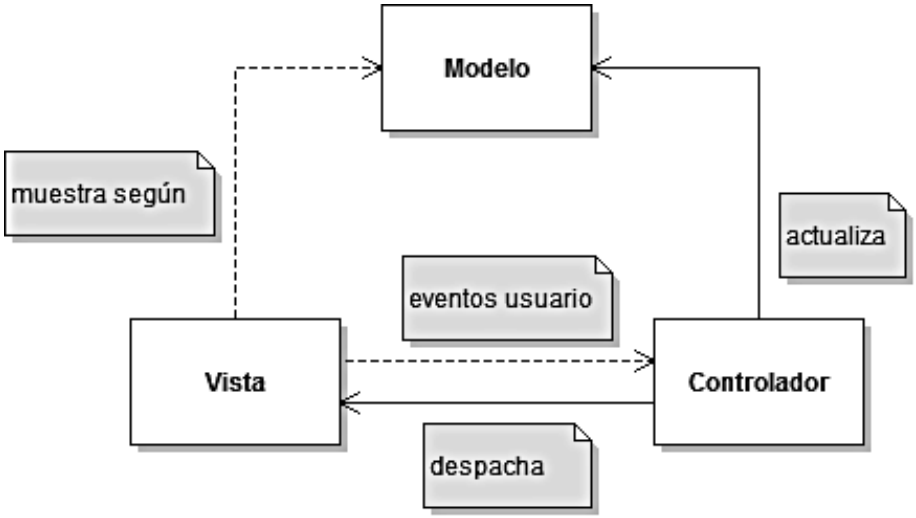
Arquitectura de *software*

CAPÍTULO 5

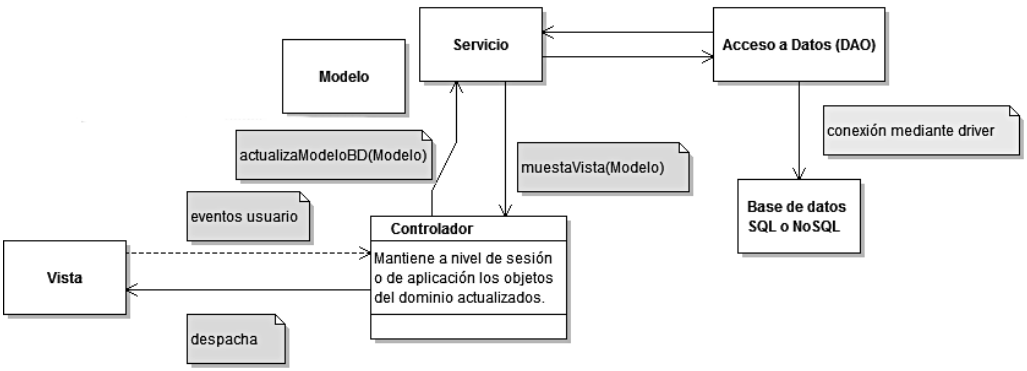
Al igual que ocurre con los edificios, las aplicaciones deben diseñarse a alto nivel para posteriormente ir definiendo el diseño de las partes o módulos que componen la aplicación. A este diseño de alto nivel lo denominamos *arquitectura de software*. La arquitectura de *software* no se centra en elementos tales como la tipología de las entradas del usuario, ni se centra en el color de las ventanas de la aplicación. Digamos que esta parte es más propia de un especialista en diseño gráfico o experiencia de usuario que de un arquitecto de *software*. La arquitectura se centra en diseñar a grandes rasgos cómo serán las partes de la aplicación y la unión de todas estas partes para conseguir un sistema completo. Un arquitecto de *software* puede seguir diseñando los módulos que ha pensado en refinamientos sucesivos (bajando de nivel). Pero la arquitectura compone la unión de grandes bloques que denominamos *capas de la aplicación*.

5.1 Patrón MVC

Para definir la arquitectura de una aplicación, lo común es encontrarnos con el patrón de diseño arquitectónico MVC. MVC es una abstracción de casi cualquier arquitectura, en la que el acceso coherente a datos se define como *modelo*, el corazón de la aplicación (la que gestiona todo) es el *controlador* y aquello que ve el usuario (las páginas web o las pantallas de las aplicaciones) se define como la *vista*. Aquí aparece un pequeño esquema explicativo de cómo funciona dicho patrón.



Este esquema, como hemos dicho, suele ser una abstracción, para cualquier lenguaje y para cualquier arquitectura. En una aplicación empresarial Java, lo normal es que el modelo de la aplicación esté desarrollado bajo la API de EJB. El controlador está compuesto por *servlets* que responden a las peticiones de los usuarios; siendo las vistas las páginas JSP que se lanzan al navegador. Pero la arquitectura, sobre todo en el lado del *back-end*, suele ser más compleja. Lo normal, para una aplicación Java, es tener las cinco capas que se muestran en la imagen siguiente:



Vemos que el modelo se convierte en el conjunto de clases que sirven para comunicar el controlador con la vista y la capa de servicio. En la capa de servicio comienza el *back-end* de la aplicación, donde las conexiones de datos ya serán con objetos propios de la base de datos.

Previamente, en nuestro texto, hemos visto cómo instalar un servidor de aplicaciones (en concreto Oracle WebLogic Server, pero podría haber sido cualquier otro). La filosofía de trabajo con servidores de aplicaciones viene a definir lo que denominamos el *desarrollo basado en componentes*.

Según Wikipedia "un componente de *software* individual es un paquete de *software*, un servicio web o un módulo que encapsula un conjunto de funciones relacionadas (o de datos)".

En nuestro discurso (genérico, pero orientado a JEE), digamos que un componente de *software* es una unidad de *software* que tiene un alto nivel de cohesión (porque las partes del componente realizan una única tarea concreta) y un bajo nivel de acoplamiento (porque es fácil modificar la estructura interna del componente), ya que las partes están claramente diferenciadas (siendo, por lo tanto, fácil de mantener). Todo componente de *software* que se precie debe ser modular (módulos o partes internas bien diferenciadas) y cohesivo (realizan una tarea concreta lo mejor posible). Los componentes de *software* en la JEE se definen por medio de la API de EJB.

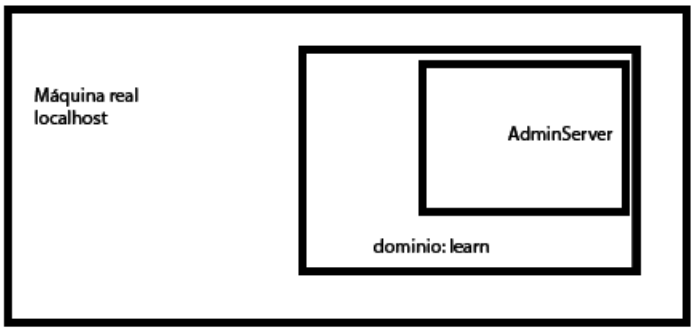
5.2 Arquitectura de *software* y servidor de aplicaciones

Enterprise JavaBeans es una API de JEE de acceso a la lógica de negocio de las aplicaciones. Dicha API en su versión 3.0 está especificada en la JSR 220. El objetivo de la API es ofrecer una colección de objetos del lado del servidor. Estos objetos son los denominados *EJB*. Lo que se pretende con los EJB es que los programadores puedan abstraerse de problemas genéricos de una aplicación empresarial (tales como concurrencia, persistencia, seguridad...). Todo ello basado en una arquitectura de componentes (cada EJB es un componente). La API de EJB hace uso de un conjunto de tecnologías:

- **RMI** (*Remote Method Invocation* o Invocación Remota de Métodos).
- **JNDI** (*Java Naming and Directory Interface* o Interfaz de Directorio y Nombrado de Java). Esta tecnología la usaremos para localizar los EJB, las conexiones y otros componentes dentro de la estructura de directorios del servidor de aplicaciones.
- **CORBA** (*Common Object Request Broker Architecture*). Para la comunicación remota. Es una tecnología que permite que diversos componentes escritos en diferentes lenguajes de programación y desplegados en distintas máquinas puedan acoplarse para desarrollar una aplicación empresarial.
- **JMS** (*Java Message Service* o Servicio de Mensajería de Java). Permite la comunicación asíncrona entre componentes, mediante un sistema de eventos. Los publicadores de eventos los envían a una cola punto a punto o lo suscriben a un tema.

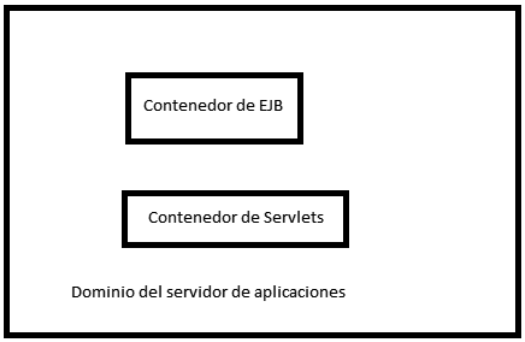
No va a ser objetivo de nuestro manual estudiar CORBA, pero estudiaremos de forma aplicada las otras tecnologías.

En el capítulo anterior hemos instalado un servidor de aplicaciones. Pues ahora comenzaremos a hacer uso de él. El servidor de aplicaciones está basado en la lógica de contenedores, en la que el contenedor se encarga de gestionar los problemas genéricos de una aplicación empresarial (tales como concurrencia o seguridad). Recordemos el diagrama del servidor de aplicaciones que hemos instalado:

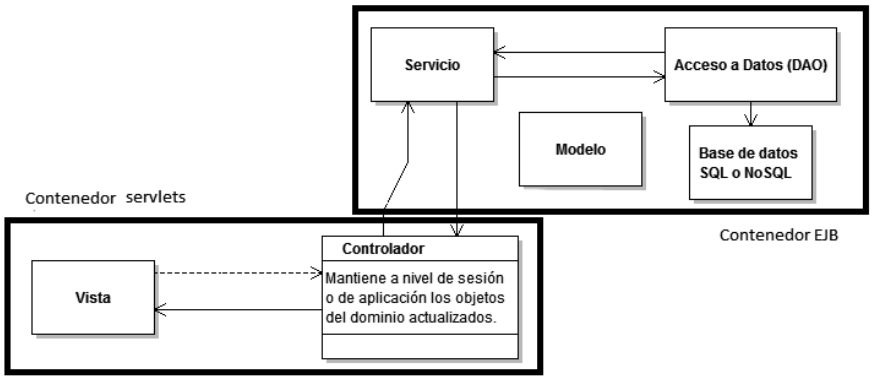


Dentro de nuestro dominio, en el servidor que hayamos configurado o haya configurado el instalador (en este caso "AdminServer"), habrá dos contenedores: el contenedor de EJB y el contenedor de *servlets*. El contenedor de EJB se va a encargar de gestionar todos los componentes EJB que residan en el servidor, mientras que el contenedor de *servlets* se encargará de escuchar las peticiones web que se reciban mediante HTTP (por método GET o POST) para gestionar la respuesta del sistema a las peticiones de los usuarios e implementar el controlador de la aplicación, despachando las páginas JSP que se precisen en cada momento. Actualmente están de moda los términos *front-end* y *back-end*. El *front-end* hace referencia a la presentación de la aplicación y lo podríamos entender como la unión de la capa de servicio con la interfaz de usuario gracias al controlador. El *back-end* hace referencia a las capas de *software* existentes desde la base de datos hasta que ofrecemos una API coherente como capa de servicio al *front-end* de nuestras aplicaciones.

Como hemos dicho, en el dominio tendremos dos contenedores: uno de EJB *(back-end)* y otro de *servlets (front-end)*. Pero no olvidemos nunca que el código que escribamos en un EJB, en un *servlet* o en una página JSP es código de servidor, distinto al código de cliente que enviaremos a los dispositivos de los usuarios en el lenguaje JavaScript.



Una vez visto el esquema básico de la arquitectura de *software* de una aplicación, veamos cómo se desplegarían en nuestro servidor de aplicaciones.



Los objetos del modelo no son en sí EJB. Son objetos serializables (implementan la interfaz serializable) que permiten el paso de parámetros entre el controlador y la capa de servicio. A su vez los podemos usar como JavaBeans para ser gestionados por la capa de presentación (la vista).

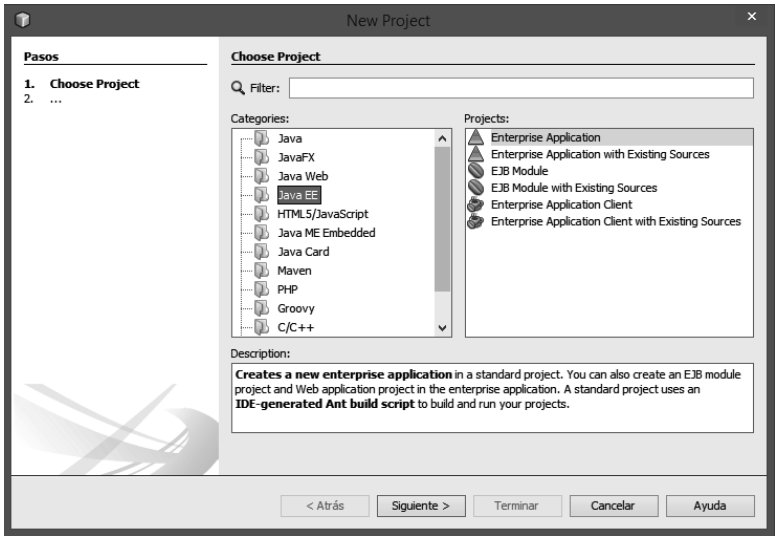
Acudamos de nuevo a Wikipedia para leer la definición de *serialización*: "la serialización (o *marshalling* en inglés) consiste en un proceso de codificación de un objeto en un medio de almacenamiento (como puede ser un archivo o un *buffer* de memoria)". Afortunadamente, Java nos aísla de este problema de bajo nivel haciendo que nuestros objetos que puedan ser enviados o recibidos a través de la red o a otras máquinas virtuales implementen la interfaz serializable.

5.3 Capa de presentación

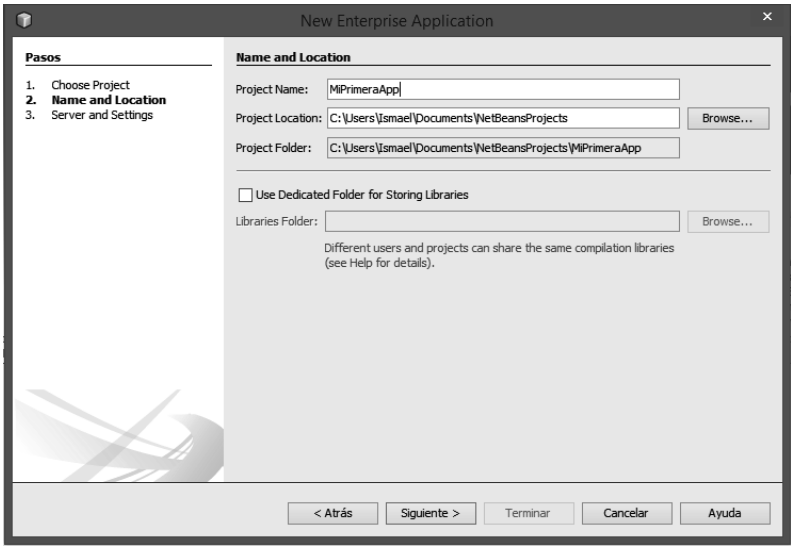
La capa de presentación también puede ser una abstracción de la arquitectura de la aplicación. Podemos entender por *capa de presentación* únicamente a los artefactos de código fuente que verán los usuarios (las vistas en sí, por ejemplo, las páginas JSP). Pero también podemos llamar *capa de presentación* al conjunto de módulos que forman las vistas, el controlador y la interfaz proporcionada por la capa de servicio (*front-end* completo). Si lo entendemos de esta segunda forma (y debemos hacerlo así), aparecen dos tecnologías que tendrán su propio contenedor (el de *servlets*). Dichas tecnologías son los *servlets* y las páginas JSP.

5.3.1 *Servlets*

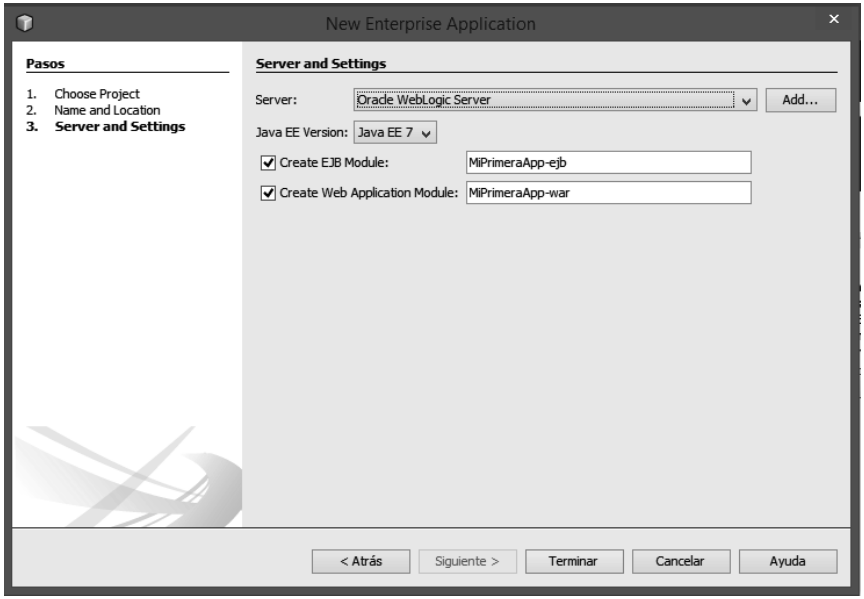
Los *servlets* son la tecnología que implementa el controlador de la aplicación respondiendo a peticiones HTTP. Como hemos visto anteriormente, un *servlet* residirá en el contenedor de *servlets* del servidor de aplicaciones. Como ya hemos visto suficiente teoría en lo que va de capítulo, vamos a comenzar con la práctica. Empecemos a definir una aplicación empresarial que contenga un *servlet*. Vayamos a NetBeans y creemos un nuevo proyecto de tipo "Aplicación Empresarial".



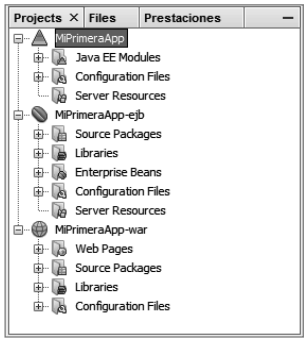
Seguidamente creamos un nuevo proyecto a desplegar en el servidor.



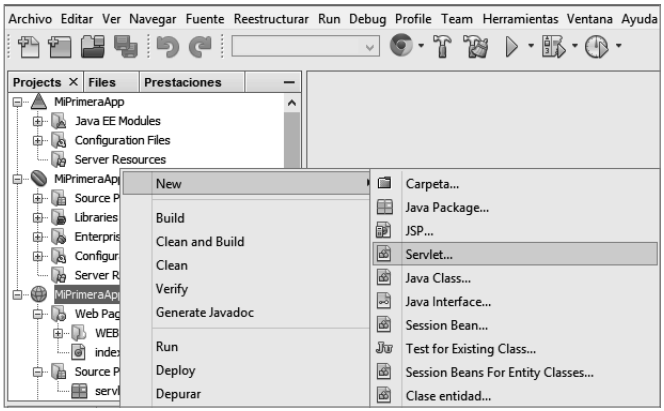
Muy importante: elegimos el servidor en el que se desplegará y la versión actual de Java EE: la 7. Podemos ver dos módulos: el de EJB y el web (cada módulo va a un contenedor).

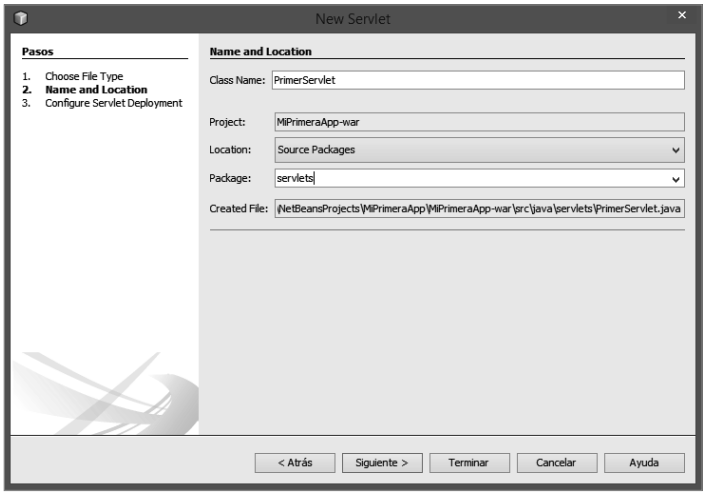


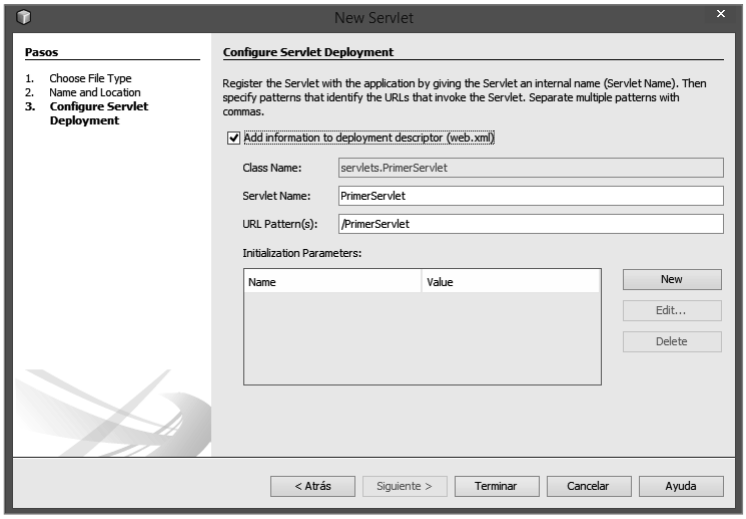
Finalmente, podremos observar en la sección de proyectos el proyecto creado (con sus dos módulos).



Para crear nuestro primer *servlet* hemos de ir al proyecto WAR, crear un paquete y dentro de él un *servlet*. Acto seguido nos preguntará si queremos añadirlo al servidor de despliegue. La respuesta es afirmativa, ya que los *servlets* no los trabajaremos con anotaciones como los EJB.







Veamos ahora el código del *servlet* que hemos creado:

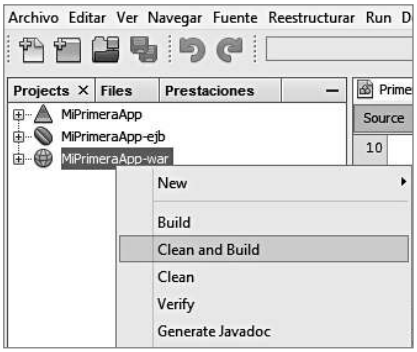
```
package servlets;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class PrimerServlet extends HttpServlet {
   protected void processRequest(HttpServletRequest request,
           HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      try (PrintWriter out = response.getWriter()) {
         out.println("<!DOCTYPE html>");
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet PrimerServlet</title>");
         out.println("</head>");
         out.println("<body>");
         out.println("<h1>Servlet PrimerServlet at " +
                  request.getContextPath() + "</h1>");
         out.println("</body>");
         out.println("</html>");
      }
   }
```

```
    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException {
      processRequest(request, response);
    }
    @Override
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException {
      processRequest(request, response);
    }
    @Override
    public String getServletInfo() {
      return "Short description";
    }
}
```

Del código podemos hacer varias apreciaciones:

- La clase "PrimerServlet" hereda de la clase "HttpServlet".
- Todas las clases referenciadas se incluyen en el paquete "javax" (contiene las API de la JEE).
- Tenemos dos métodos: "doGet" y "doPost". Ambos para atender las peticiones. Unas mediante HTTP GET y otras mediante HTTP POST.
- Ambos métodos tratan dos parámetros: la petición del cliente y la respuesta que se le envía al cliente.
- Ambos métodos llaman al método "processRequest", que es el método que procesa la petición HTTP.
- Ambos métodos lanzan las excepciones "ServletException" e "IOException".
- La salida al usuario se realiza mediante el objeto *out* que lo obtenemos con la instrucción: "PrintWriter out = response.getWriter()".

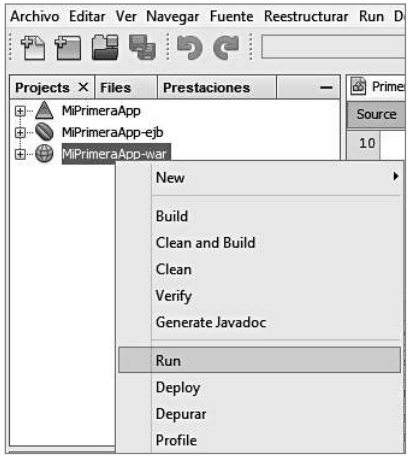
Para echar a andar nuestra aplicación, hemos de compilarla y desplegarla en el servidor. Para ello, la compilamos mediante la opción "Construir" del IDE (en nuestro caso "Clean and Build"). Nota: hemos de hacerlo sobre el proyecto web.



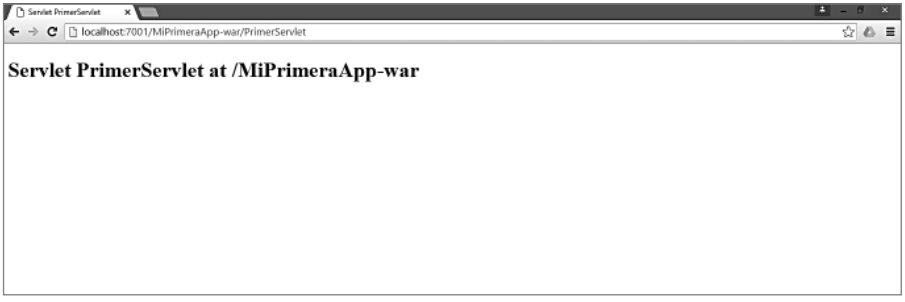
La salida del compilador es:

BUILD SUCCESSFUL (total time: 5 seconds).

Para desplegarlo y ejecutarlo, seleccionamos la opción ejecutar ("Run").



Y vemos la salida de la aplicación acudiendo a nuestro navegador y escribiendo la siguiente URL: "http://localhost:7001/MiPrimeraApp-war/PrimerServlet".



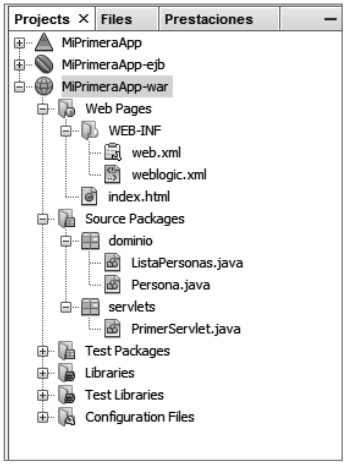
Cuando tengamos nuestros componentes EJB creados, en vez de compilar y ejecutar sobre el proyecto web, lo haremos sobre el proyecto completo (módulos EJB y WAR).

Por último, veamos el código del descriptor de despliegue del módulo web ("web.xml" dentro de la carpeta "WEB-INF"):

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <servlet>
    <servlet-name>PrimerServlet</servlet-name>
    <servlet-class>servlets.PrimerServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>PrimerServlet</servlet-name>
    <url-pattern>/PrimerServlet</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
</web-app>
```

Podemos personalizar la aplicación escribiendo en la salida (mediante el objeto *out*) lo que queramos.

Imaginemos que tenemos un listado de tres personas. De cada persona únicamente guardamos su nombre. Como es un diagrama de clases muy simple únicamente vamos a mostrar el código. Todo el código lo escribiremos en el módulo WAR. Veamos cómo queda la estructura del proyecto (mostrando la localización del descriptor de despliegue).



Hemos creado un paquete llamado *dominio*. Dicho paquete incluye la lista de personas.

- Clase "Persona.java":

```
package dominio;
public class Persona {
   private String nombre;

   public String getNombre() {
      return nombre;
   }
   public void setNombre(String nombre) {
      this.nombre = nombre;
   }
}
```

- Clase "ListaPersonas.java":

```
package dominio;
import java.util.ArrayList;
import java.util.List;
public class ListaPersonas {
   private List<Persona> personas;
   public ListaPersonas() {
      this.personas = new ArrayList<Persona>();
   }
   public List<Persona> getPersonas() {
      return personas;
   }
   public void setPersonas(List<Persona> personas) {
      this.personas = personas;
   }
}
```

- *Servlet*:

```
package servlets;
import dominio.ListaPersonas;
import dominio.Persona;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
public class PrimerServlet extends HttpServlet {
   private ListaPersonas listaPersonas;
   protected void processRequest(HttpServletRequest request,
           HttpServletResponse response)
       throws ServletException, IOException {
      this.cargarPersonas();
      response.setContentType("text/html;charset=UTF-8");
      try (PrintWriter out = response.getWriter()) {
         out.println("<!DOCTYPE html>");
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet PrimerServlet</title>");
         out.println("</head>");
         out.println("<body>");
         out.println("<h1>Servlet PrimerServlet at " +
                 request.getContextPath() + "</h1>");
         int i = 1;
         List<Persona> personas = this.listaPersonas.getPersonas();
         for(Persona p : personas) {
            out.println("<h2>Persona "+i+": "+p.getNombre()+".</h2>");
            i++;
         }
         out.println("</body>");
         out.println("</html>");
      }
   }

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

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

   @Override
   public String getServletInfo() {
      return "Short description";
   }
```

```
  private void cargarPersonas() {
    this.listaPersonas = new ListaPersonas();
    Persona p = new Persona();
    p.setNombre("Ismael");
    this.listaPersonas.getPersonas().add(p);
    p = new Persona();
    p.setNombre("Antonio");
    this.listaPersonas.getPersonas().add(p);
    p = new Persona();
    p.setNombre("Javier");
    this.listaPersonas.getPersonas().add(p);
  }
}
```

Y la salida en el navegador es la esperada:

5.3.2 Páginas JSP

Hemos visto que mediante la tecnología de *servlets* podemos crear contenido web dinámico (haciendo uso de "PrintWriter out = response.getWriter())". Pero se nos antoja un poco tedioso escribir de esta forma nuestras aplicaciones. Lanzando la salida en los *servlets*, debemos escribir el código HTML en cadenas de texto. Para solucionar este tedio se pensó en la tecnología de páginas JSP. Una página JSP es una página con sintaxis HTML que puede incorporar código Java en forma de *scriptlet* para generar el contenido dinámico. JSP es el acrónimo de *Java Server Pages* (Páginas del Servidor de Java), siendo la última versión la 2. Una vez analizada la página JSP, su información se añadirá al *servlet*.

Para poder incorporar el código Java dentro de la página JSP, se hace uso de la siguiente sintaxis:

- Expresión: "<%= ... %>". Con dicho fragmento le decimos al intérprete de JSP que debe escribir en la salida lo que indique la expresión. Indicar que la expresión no debe acabar en punto y coma (;).
- *Scriptlet*: "<% ... %>". Dentro del *scriptlet* podemos incluir cualquier código Java.
- Declaraciones: "<%! ... %>". Sirve para declarar nuevas variables en la página JSP.

Pero para poder hacer uso de las variables de nuestra aplicación, necesitamos incluir estas variables en forma de JavaBeans. No confundamos los JavaBeans con los Enterprise JavaBeans. Los Enterprise JavaBeans son los componentes que residirán en el contenedor de EJB en forma EJB, mientras que los JavaBeans son los objetos del modelo que usarán las vistas para mostrar la información.

Llegados a este punto, es bueno indicar que vamos a tener dos tipos de JavaBeans:

- De sesión.
- De aplicación.

Los *beans* de sesión serán aquellos objetos que sólo tendrán vida a lo largo de la sesión del usuario. Del mismo modo, tendremos una instancia por cada usuario que esté logado en el sistema. Los *beans* de aplicación existirán durante todo el tiempo de vida de la aplicación y tendremos una instancia para todos los usuarios del sistema. La variable de sesión será de tipo "javax.servlet.http.HttpSession", mientras que la variable de aplicación será de tipo "javax.servlet.ServletContext". Ambas variables funcionarán a modo de mapa *(HashMap)*, en el que setearemos los objetos de cada ámbito con el identificador que queramos. Lo normal es ponerle a los objetos como identificador su mismo nombre. Veamos nuestro primer *servlet* en el que se obtiene acceso a las variables de aplicación y de sesión:

```
package servlets;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class PrimerServlet extends HttpServlet {
  protected void processRequest(HttpServletRequest request,
          HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession();
    ServletContext context = getServletContext();
    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
```

```
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet PrimerServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet SegundoServlet at " + request.getContextPath() + "</h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }

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

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

    @Override
    public String getServletInfo() {
        return "Información del servlet";
    }
}
```

De este modo podremos setear en las variables *session* y *context* nuestras variables de sesión y de aplicación respectivamente. Creemos un segundo *servlet* en el que seteemos nuestras variables con dos cadenas (una cadena a la variable de sesión y otra cadena a la variable de aplicación):

```
package servlets;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
```

```
public class SegundoServlet extends HttpServlet {
   private String variableSesion = "cadena 1";
   private String variableAplicacion = "cadena 2";
   protected void processRequest(HttpServletRequest request,
           HttpServletResponse response)
       throws ServletException, IOException {
     HttpSession session = request.getSession();
     ServletContext context = getServletContext();
     if (session.getAttribute("variableSesion") == null) {
       session.setAttribute("variableSesion", variableSesion);
     }
     if (context.getAttribute("variableAplicacion") == null) {
       context.setAttribute("variableAplicacion", variableAplicacion);
     }
     response.setContentType("text/html;charset=UTF-8");
     try (PrintWriter out = response.getWriter()) {
       out.println("<!DOCTYPE html>");
       out.println("<html>");
       out.println("<head>");
       out.println("<title>Servlet PrimerServlet</title>");
       out.println("</head>");
       out.println("<body>");
       out.println("<h1>Servlet SegundoServlet at " + request.getContextPath() + "</h1>");
       out.println("</body>");
       out.println("</html>");
     }
   }

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

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

   @Override
   public String getServletInfo() {
     return "Información del servlet";
   }
}
```

De esta forma estamos asociando una cadena a la sesión y otra cadena a la aplicación. Pero lo normal será tener seteados objetos del modelo del dominio. Y estos objetos del dominio son los conocidos JavaBeans. Dichos objetos son accesibles desde las páginas JSP. Llamamos a un objeto JavaBean si cumple ciertas condiciones:

- Tienen un constructor sin argumentos.
- Implementan la interfaz "java.io.Serializable".
- Tienen los atributos privados.
- Accedemos a ellos a través de *getters & setters*. Para los booleanos los *getters* comienzan con la palabra *is*.

Para poder hacer uso de ellos en las páginas JSP debemos hacer uso de la directiva *page* y de la acción de "jsp useBean".

Las directivas de JSP son *page*, *taglib* e *include*. Con las directivas le indicamos al motor de JSP información que afectará a la estructura del *servlet* generado (en tiempo de compilación). La sintaxis de una directiva es "<%@directiva propiedad="valor">".

La directiva *page* le indica al motor JSP cierta información de la página que se está tratando. Por ejemplo, le indicará el tipo de contenido de la página, si tendrá acceso a la sesión o no, el listado de paquetes a importar...

La directiva *taglib* sirve para importar conjuntos de etiquetas que sean interpretables por el motor de JSP. Una librería de etiquetas muy conocida es JSTL.

La directiva *include* le indica al motor de JSP que debe incluir en la página la información del fichero indicado en la URL. Dicha inclusión se realiza en tiempo de compilación y podemos incluir el contenido de otra página JSP.

Las acciones de JSP se lanzan en tiempo de ejecución y tienen diversos cometidos como la transferencia de control, la inclusión de objetos o la inclusión de páginas (pero en tiempo de ejecución). Comencemos mostrando el uso de la acción "useBean".

Después de haber realizado la lectura somera de la teoría sobre JSP, vamos a reescribir nuestro ejemplo del listado de personas usando ahora la tecnología de JSP. Reescribimos los *beans* para que implementen la interfaz "Serializable", con constructores sin argumentos y refactorizamos el *servlet* "PrimerServlet".

Los *beans* "Persona" y "ListaPersonas" quedan como siguen:

- Clase "Persona.java":

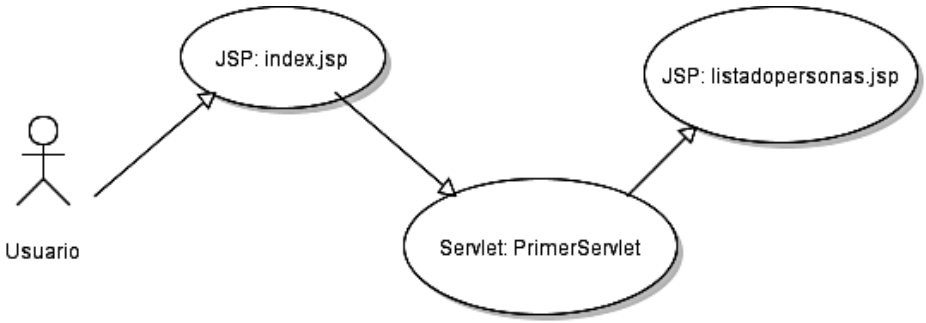
```
package dominio;
import java.io.Serializable;
public class Persona implements Serializable {
  public Persona(){}
  private String nombre;
```

```
   public String getNombre() {
      return nombre;
   }
   public void setNombre(String nombre) {
      this.nombre = nombre;
   }
}
```

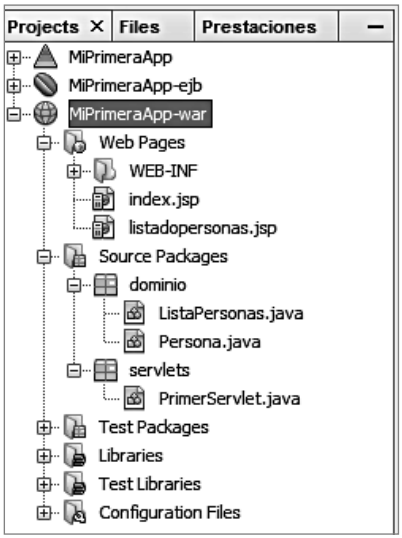
- Clase "ListaPersonas.java":

```
package dominio;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
public class ListaPersonas implements Serializable {
   private List<Persona> personas;
   public ListaPersonas() {
      this.personas = new ArrayList<Persona>();
   }
   public List<Persona> getPersonas() {
      return personas;
   }
   public void setPersonas(List<Persona> personas) {
      this.personas = personas;
   }
}
```

Pensemos en un modelo en el cual accedemos a una página principal escrita en JSP ("index.jsp"). Una vez que hayamos accedido a dicha página, mediante un botón de formulario accederemos al *servlet*. Una vez en el *servlet*, se creará el listado de personas y se seteará la variable "listaPersonas" como variable de aplicación. Posteriormente despacharemos en la respuesta la página "listadopersonas.jsp". El diagrama de funcionamiento podría ser el siguiente:



Veamos ahora cuál es la estructura del proyecto:



- Clase "PrimerServlet.java":

```
package servlets;
import dominio.ListaPersonas;
import dominio.Persona;
import java.io.IOException;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class PrimerServlet extends HttpServlet {
   private ListaPersonas listaPersonas;
   protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
         throws ServletException, IOException {
      // Sesión. No la usamos en este caso. Para los beans de sesión.
      HttpSession session = request.getSession();
      // Aplicación. Para los beans de aplicación.
      ServletContext context = getServletContext();
      this.cargarPersonas();
      if (context.getAttribute("listaPersonas") == null) {
         context.setAttribute("listaPersonas", this.listaPersonas);
      }
      response.sendRedirect("listadopersonas.jsp");
   }
```

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

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

    @Override
    public String getServletInfo() {
      return "Información del servlet";
    }

    private void cargarPersonas() {
      this.listaPersonas = new ListaPersonas();
      Persona p = new Persona();
      p.setNombre("Ismael");
      this.listaPersonas.getPersonas().add(p);
      p = new Persona();
      p.setNombre("Antonio");
      this.listaPersonas.getPersonas().add(p);
      p = new Persona();
      p.setNombre("Javier");
      this.listaPersonas.getPersonas().add(p);
    }
}
```

- Página JSP "index.jsp":

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Página JSP</title>
  </head>
  <body>
    <p>Sigue el enlace para ver el listado de personas...</p>
    <form action="PrimerServlet">
```

```
        <input type="submit" name="Enviar" />
      </form>
    </body>
</html>
```

- Página JSP "listadopersonas.jsp":

```
<%@page contentType="text/html" pageEncoding="UTF-8"
            import="java.util.*,dominio.*" %>
<jsp:useBean id="listaPersonas" class="dominio.ListaPersonas" scope="application" />
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Listado de personas</title>
  </head>
  <body>
    <h1>Listado de personas</h1>
    <%
      List<Persona> personas = listaPersonas.getPersonas();
      for(Persona p : personas) {
    %>
          <h2>
            <%=p.getNombre()%>
          </h2>
    <%
      }
    %>
  </body>
</html>
```

Podemos ver que en la propiedad *action* del formulario podemos indicar únicamente el nombre del *servlet*. También podemos ver como hemos usado el *bean* de aplicación mediante la acción "<jsp:useBean ... />". Mediante la directiva *page* hemos indicado el tipo de fichero de la salida, su codificación de caracteres y los paquetes que debe importar (para poder hacer uso de las clases "List" y "Persona"). Veamos la salida en el navegador:

5.4 Enterprise JavaBeans

Nos centraremos ahora en la tecnología de EJB para ver cómo podemos implementar el *back-end* de nuestras aplicaciones de forma profesional (mediante componentes). Estudiaremos los beneficios de usar EJB así como la implementación de cada tipo de ellos.

Como hemos explicado anteriormente, los componentes que van a residir en nuestro servidor de aplicaciones van a tener la forma de EJB. Estos componentes deben ofrecer en forma de contrato una interfaz para aquellas otras aplicaciones o componentes que quieran usarlos. En este punto debemos nombrar que existen dos tipos de interfaces (si hablamos de la JEE):

- Interfaz local.
- Interfaz remota.

La interfaz local es aquella que usarán los componentes que están situados en el mismo servidor de aplicaciones. La interfaz remota es aquella que usarán los componentes que estén en servidores de aplicaciones diferentes, o aquellos fragmentos de código que, aun ejecutándose en la misma máquina que el servidor, no estén desplegados en éste. Los componentes susceptibles de ser enviados a otros servidores deben ser serializables.

Los EJB son componentes de *software*. Son accesibles desde el propio servidor de aplicaciones o desde otros servidores (haciendo uso de RMI). Nosotros los usaremos haciendo uso de la interfaz local o mediante la interfaz remota.

Desde la implementación de EJB 3.0, siendo la actual la implementación 3.2, es muy simple desplegar los *beans* en los contenedores, ya que no hace falta crear descriptores de despliegue en XML, sino que lo realizamos por medio de anotaciones en el código de nuestras clases. Por ejemplo, la interfaz local de un *bean* de sesión:

```
package sessionbeans;
import javax.ejb.Local;
@Local
public interface ISessionBeanLocal {
  // Métodos de la interfaz local.
}
```

Del mismo modo, una interfaz remota sería la siguiente:

```
package sessionbeans;

import javax.ejb.Remote;

@Remote
public interface ISessionBeanRemote {
  // Métodos de la interfaz remota.
}
```

Los componentes de *software* son cada una de las piezas del puzle en la arquitectura orientada a servicios. Esta arquitectura no es más que un paradigma que permite el desarrollo de sistemas altamente escalables y distribuidos en los que se facilita la comunicación, el mantenimiento y la integración de aplicaciones menores.

Tenemos varios tipos de EJB:

- **De entidad**.
  - Persistencia gestionada por el contenedor.
  - Persistencia gestionada por el *bean*.
- **De sesión**.
  - Sin estado: *stateful*.
  - Con estado: *stateless*.
  - Instancia única: *singleton*.
- **Orientados a mensajes.** Hacen uso de la JMS.
  - De cola: *queue*.
  - Tema: *topic*.

5.4.1 *Beans* de entidad. Acceso usando JPA

Los *beans* de entidad *(entity)* son los relacionados con la persistencia de objetos en nuestra base de datos. Usando la *Java Persistence API* (API de Persistencia de Java) en su versión 2.1, las entidades serán los equivalentes directos a las tablas de nuestra base de datos. En el capítulo de instalación de Oracle WebLogic Server vimos cómo conectar una base de datos con el servidor. Nuestra base de datos de ejemplo se llamaba *learndb*.

La Java Persistence API es un estándar que debe ser implementado (definido mediante JSR). Existen varias implementaciones de la JPA, entre ellas (recogido de Wikipedia):

- Hibernate.
- ObjectDB.
- TopLink.
- CocoBase.
- EclipseLink.
- OpenJPA.
- Kodo.
- DataNucleus.
- Amber.

En nuestro aprendizaje usaremos la versión de EclipseLink porque es la más fácil de usar en nuestro entorno servidor (el servidor de aplicaciones ya la trae instalada). Usando JPA nos olvidamos de la estructura relacional de la base de datos. Cada *bean* de entidad irá precedido de la anotación "@Entity". Sólo nos tendremos que preocupar de definir el modelo de la base de datos a modo de diagrama de clases. Esto se consigue gracias al ORM (patrón de mapeo objeto-relacional) que convierte los atributos entidades en claves ajenas.

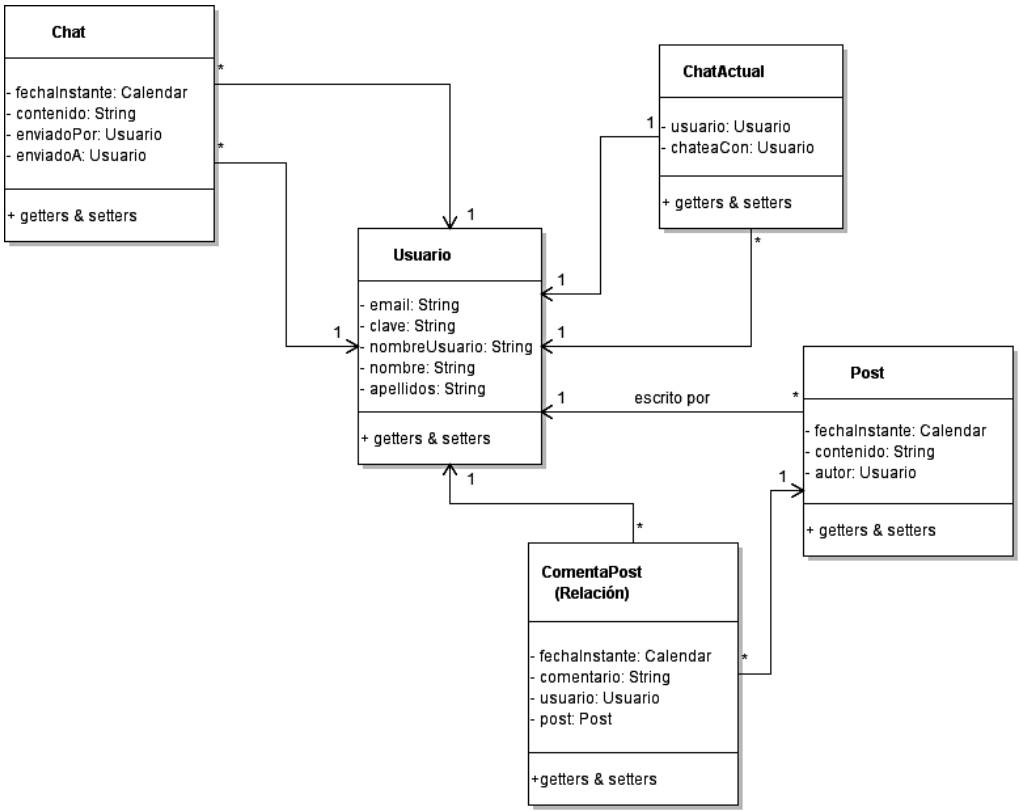
Para modelar las relaciones en JPA, usamos las siguientes anotaciones en los atributos:

- @OneToOne: uno a uno.
- @OneToMany: uno a muchos.
- @ManyToOne: muchos a uno.
- @ManyToMany: muchos a muchos (intentaremos evitarlas).

Comencemos con la práctica. Como ejemplos y ejercicios a lo largo del libro se va a proponer crear una aplicación web completa usando JEE. Toda aplicación, al igual que un edificio, comienza desde los cimientos. Los cimientos en *software* son los datos. Propongamos realizar, a lo largo del texto, una aplicación consistente en un blog

que permita insertar comentarios a los *posts* de los distintos usuarios registrados. Del mismo modo, se podrá chatear particularmente con cada uno de los usuarios.

El modelo de datos que se propone para este ejercicio es el siguiente (definido como *diagrama de clases*):



En dicho ejercicio tenemos cinco entidades:

- "Usuario".
- "Post".
- "ComentaPost".
- "Chat" (un mensaje de chat).
- "ChatActual". Para ver quién está hablando particularmente con quién en cada momento.

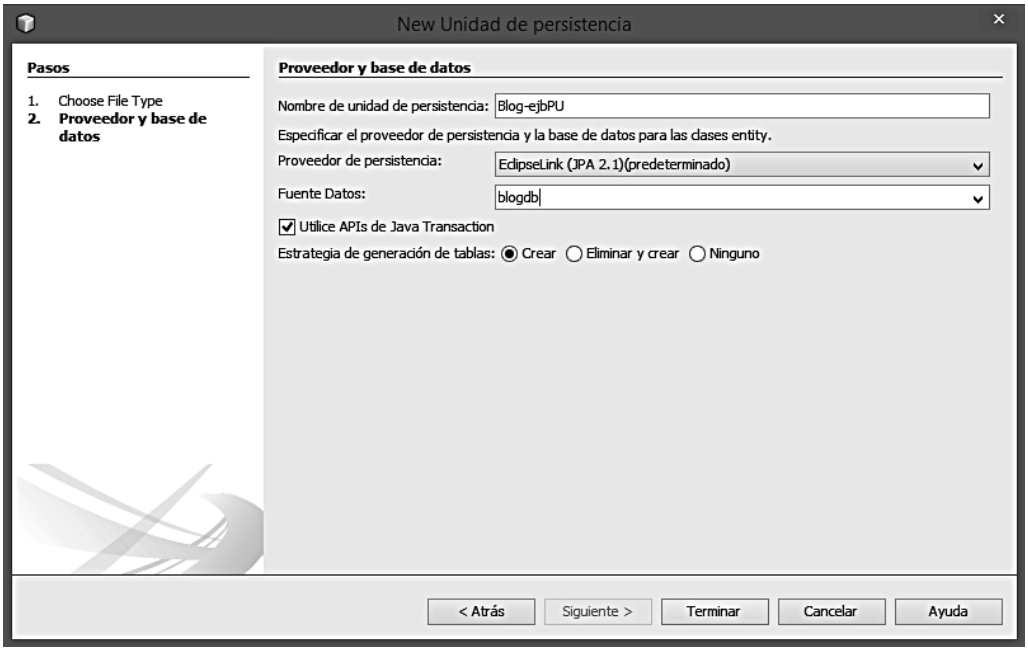
Las relaciones "muchos a muchos" las hemos modelado como una nueva entidad ("Chat" y "ComentaPost"). Como regla general evitaremos las relaciones "muchos a muchos" en el diseño, ya que pueden acarrear problemas si no tenemos mucha pericia en el uso de estas relaciones.

La relación "ChatActual" es una relación "uno a muchos" con atributos (con el atributo "Usuario"). Por ello mismo es necesario definirla como una nueva entidad.

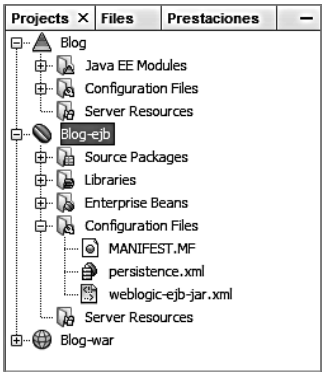
Comencemos con la práctica. Creemos un nuevo proyecto de aplicación empresarial en el que tengamos los módulos que se desplegarán: EJB y WAR. El proyecto se llamará *Blog*. El módulo en el que escribiremos nuestros *beans* de entidad (y todos los demás EJB) será el módulo EJB. Para poder persistir los objetos, debemos crear una nueva unidad de persistencia en nuestro proyecto. Haciendo clic con el botón derecho sobre el módulo EJB, podremos crear los distintos elementos que iremos estudiando.

Creemos una base de datos denominada *Blog* en nuestro gestor MySQL del mismo modo que creamos la base de datos "learn" en el capítulo de instalación del servidor. El nombre que le asignaremos en el servidor de aplicaciones será "JDBCMySQLBlog". El nombre JNDI será "blogdb".

Creando la nueva unidad de persistencia. Podemos observar que el proveedor de la unidad de persistencia es EclipseLink y que la fuente de datos es "blogdb" (nombre en el árbol JNDI que le hemos asignado).



Acto seguido, vemos dónde queda ubicada la unidad de persistencia ("persistence.xml").



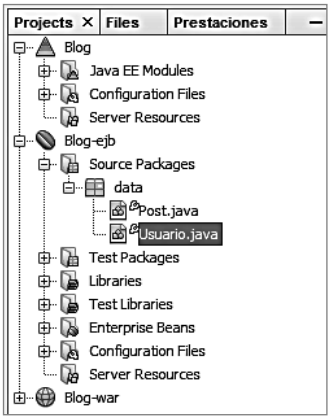
Código XML de la unidad de persistencia:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
 <persistence-unit name="Blog-ejbPU" transaction-type="JTA">
  <jta-data-source>blogdb</jta-data-source>
  <exclude-unlisted-classes>false</exclude-unlisted-classes>
  <properties>
   <property name="javax.persistence.schema-generation.database.action"
         value="create-or-extend-tables"/>
  </properties>
 </persistence-unit>
</persistence>
```

Observamos con detenimiento que hemos modificado la estrategia de generación de tablas: "create-or-extend-tables".

Acto seguido, comenzamos a crear nuestras entidades. Debemos indicar que por defecto, según el procedimiento seguido, la persistencia será gestionada por el contenedor. Podríamos indicar que la persistencia la gestionaríamos nosotros haciendo uso de JDBC. Para el lector que quiera profundizar en el tema, puede investigar sobre BMP *(Bean-Managed Persistence)*. Se considera que el manual pretende dar el conocimiento genérico sobre la JEE para desarrollar una aplicación completa de la forma más fácil y rápida.

Para nuestras entidades, crearemos un paquete llamado *data*.



Escribamos el código de las entidades "Post" y "Usuario".

- Entidad "Usuario.java":

```
package data;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Usuario implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String email;
   private String clave;
   private String nombreUsuario;
   private String nombre;
   private String apellidos;
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public String getEmail() {
      return email;
   }
```

```
public void setEmail(String email) {
   this.email = email;
}
public String getClave() {
   return clave;
}
public void setClave(String clave) {
   this.clave = clave;
}
public String getNombreUsuario() {
   return nombreUsuario;
}
public void setNombreUsuario(String nombreUsuario) {
   this.nombreUsuario = nombreUsuario;
}
public String getNombre() {
   return nombre;
}
public void setNombre(String nombre) {
   this.nombre = nombre;
}
public String getApellidos() {
   return apellidos;
}
public void setApellidos(String apellidos) {
   this.apellidos = apellidos;
}
@Override
public int hashCode() {
   int hash = 0;
   hash += (id != null ? id.hashCode() : 0);
   return hash;
}
@Override
public boolean equals(Object object) {
   if (!(object instanceof Usuario)) {
      return false;
   }
   Usuario other = (Usuario) object;
   if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
      return false;
   }
   return true;
}
@Override
public String toString() {
```

```
        return "data.Usuario[ id=" + id + " ]";
    }
}
```

- Entidad "Post.java":

```
package data;
import java.io.Serializable;
import java.util.Calendar;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
@Entity
public class Post implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private Calendar fechaInstante;
    private String contenido;
    @ManyToOne
    private Usuario autor;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public Calendar getFechaInstante() {
        return fechaInstante;
    }
    public void setFechaInstante(Calendar fechaInstante) {
        this.fechaInstante = fechaInstante;
    }
    public String getContenido() {
        return contenido;
    }
    public void setContenido(String contenido) {
        this.contenido = contenido;
    }

    public Usuario getAutor() {
        return autor;
    }
```

```
    public void setAutor(Usuario autor) {
        this.autor = autor;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Post)) {
            return false;
        }
        Post other = (Post) object;
        if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "data.Post[ id=" + id + " ]";
    }
}
```

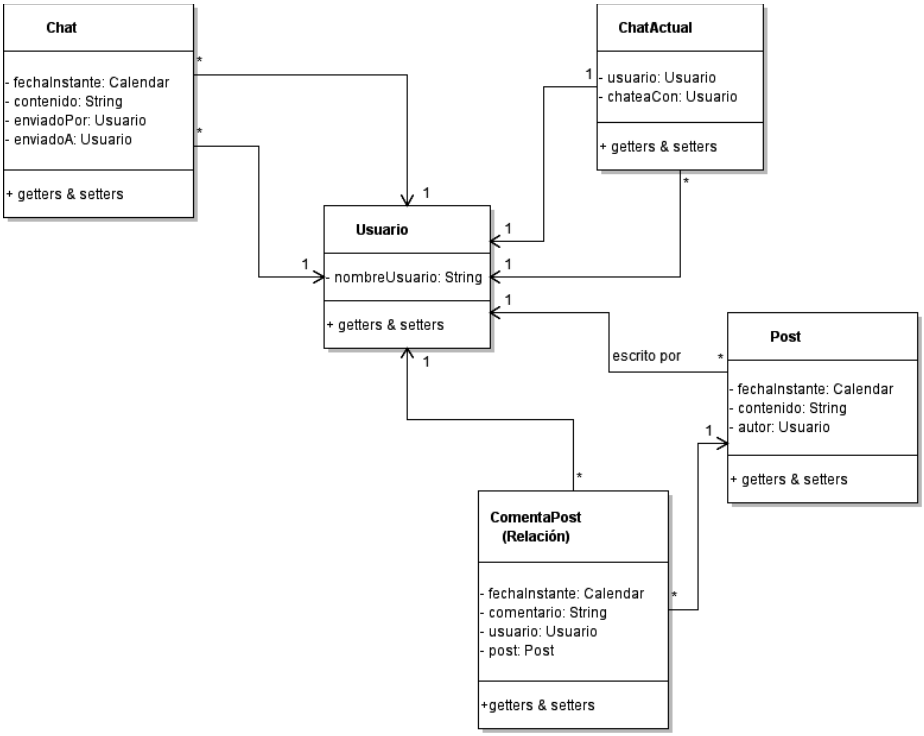
Comentarios:

- Las entidades implementan la interfaz serializable.
- Identificamos las entidades con la anotación "Entity". En EJB 3.0, desaparecen los descriptores de despliegue para los EJB.
- El campo clave será el campo "id", de tipo "Long". La estrategia de generación de claves será automática.
- El atributo "Usuario" en la entidad "Post" se define mediante la relación "muchos a uno".

Al ser la persistencia gestionada por el contenedor, podemos indicar que todas las operaciones sobre la base de datos que afecten a una sola tabla tendrán la consideración de transacción. Imaginemos que queremos insertar cien usuarios mediante un bucle. Al afectar sólo a la tabla de usuarios, el contenedor de EJB hará *rollback* si falla alguna inserción (o inserta todos los usuarios o no inserta ninguno).

5.4.2 Ejercicio 2

Dado el esquema de la base de datos del ejemplo del Blog,



y habiendo implementado durante el tema la entidad de "Post", se plantea al lector refactorizar la entidad de "Usuario", para que sólo guardemos de él el *nick* ("nombreUsuario"), y, además, terminar de implementar las entidades, en concreto:

- "ComentaPost".
- "Chat".
- "ChatActual".

5.4.3 *Beans* de sesión

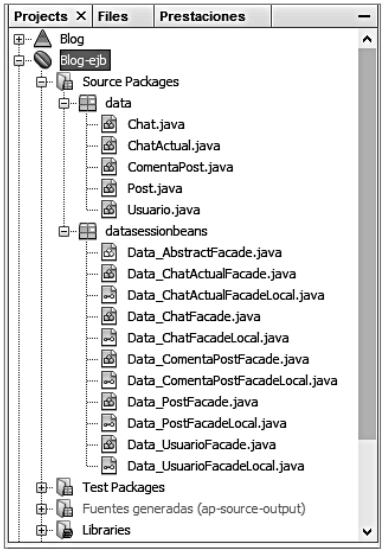
Tenemos tres tipos de *beans* de sesión:

- Sin estado: *stateless*.
- Con estado: *stateful*.
- Instancia única: *singleton*.

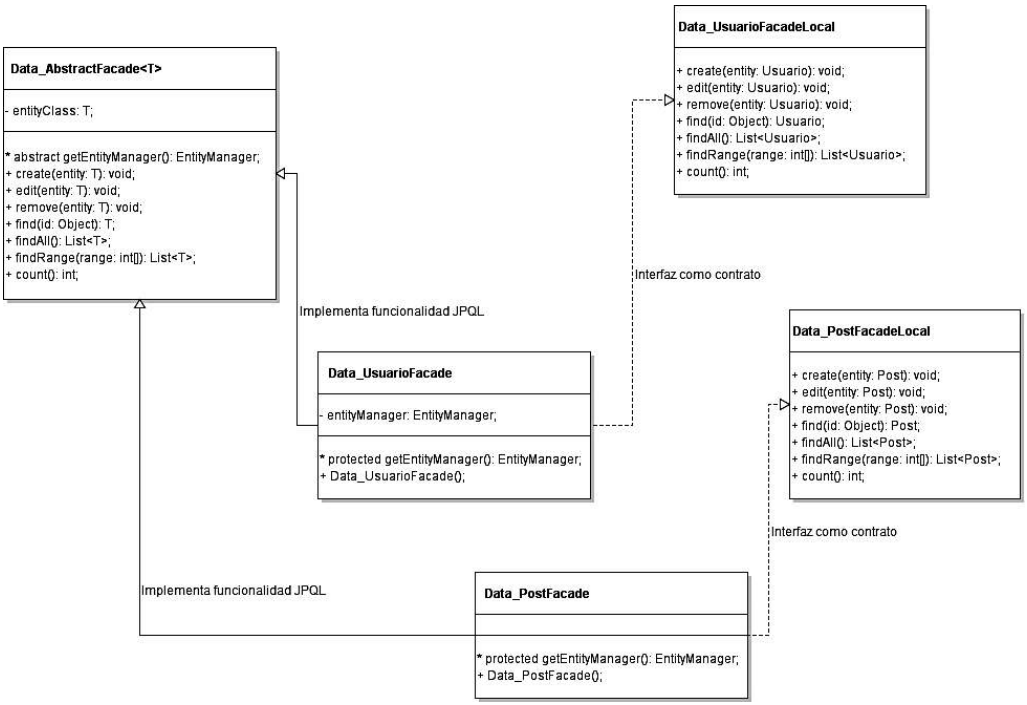
Los *beans* de sesión con estado son aquellos componentes *software* del lado del servidor que son capaces de mantener un estado conversacional con el usuario de la aplicación (es capaz de mantener el estado). Por el contrario, los *beans* de sesión sin estado están disponibles para cualquier usuario, ya que no guardan el estado conversacional con el cliente. El *singleton* es un tipo de EJB que es único para todas las peticiones sobre él (es compartido por toda la aplicación). El ejemplo más común de un *bean* de sesión con estado es el del carrito de la compra de un comercio electrónico. Cada usuario necesita almacenar en sesión su carrito. Cada objeto EJB debe ser único para cada usuario. Por el contrario, los *beans* de sesión sin estado son comunes. El ejemplo más típico es el *bean* localizador de servicios (acceso coherente al *back-end*). La utilidad del *singleton* es que nos sirve para compartir información entre toda la aplicación. ¿A qué nos recuerda esto? A los *beans* de aplicación.

5.4.3.1 Sin estado: *stateless*

Para poder realizar las operaciones del CRUD *(Create-Read-Update-Delete)* sobre las tablas de nuestra base de datos, necesitamos definir nuevos *stateless session beans* para las clases "Entity". Haciedo clic con el botón derecho sobre el módulo EJB encontraremos esta opción. A estos *beans* de sesión de bajo nivel de acceso a datos les antepondremos la palabra "Data_". Estos *beans* estarán disponibles únicamente desde el servidor de aplicaciones, por lo tanto, sólo deben implementar la interfaz local. Veamos cómo quedarían los *beans* de sesión sin estado para las operaciones del CRUD de las entidades "Usuario" y "Post". La estructura del proyecto es la siguiente:



Observamos que hemos creado una clase "Data_AbstractFacade" genérica que implementa la funcionalidad. Del mismo modo, hemos creado un par de artefactos por cada entidad: la interfaz que funciona a modo de contrato y la implementación de dicha interfaz. Para no repetir código, la funcionalidad se obtiene desde la clase "AbstractFacade", usándose el tipo de dato genérico "<T>" que permite el lenguaje Java. A continuación mostramos el diagrama de clases para las entidades "Usuario" y "Post".



La implementación de los métodos comunes se realiza con el lenguaje JPQL. Éste es un lenguaje de consulta orientado a objetos definido como parte de la JPA.

Veamos ahora el código de los "Session Beans" propuestos:

- Interfaz local a modo de contrato del *bean* "Data_UsuarioFacade". Interfaz "Data_UsuarioFacadeLocal":

```
package datasessionbeans;
import data.Usuario;
import java.util.List;
import javax.ejb.Local;
@Local
public interface Data_UsuarioFacadeLocal {
  void create(Usuario usuario);
  void edit(Usuario usuario);
```

```
  void remove(Usuario usuario);
  Usuario find(Object id);
  List<Usuario> findAll();
  List<Usuario> findRange(int[] range);
  int count();
}
```

- *Bean* de sesión sin estado "Data_UsuarioFacade.java":

```
package datasessionbeans;
import data.Usuario;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Stateless
public class Data_UsuarioFacade extends Data_AbstractFacade<Usuario> implements Data_
UsuarioFacadeLocal {
  @PersistenceContext(unitName = "Blog-ejbPU")
  private EntityManager em;
  @Override
  protected EntityManager getEntityManager() {
    return em;
  }
  public Data_UsuarioFacade() {
    super(Usuario.class);
  }
}
```

La anotación "@Stateless" define un *bean* de sesión sin estado. Podemos apreciar el uso que se hace de la unidad de persistencia (etiqueta "@PersistenceContext") para definir el objeto de la clase "EntityManager".

El *bean* de sesión anterior tiene implementados los métodos de la interfaz por heredar de la clase "Data_AbstractFacade" (como hemos visto anteriormente definimos el tipo T en la definición de la clase).

- Clase "Data_AbstractFacade.java" (uso de JPQL):

```
package datasessionbeans;
import java.util.List;
import javax.persistence.EntityManager;
public abstract class Data_AbstractFacade<T> {
  private Class<T> entityClass;
  public Data_AbstractFacade(Class<T> entityClass) {
    this.entityClass = entityClass;
  }
  protected abstract EntityManager getEntityManager();
```

```
    public void create(T entity) {
        getEntityManager().persist(entity);
    }
    public void edit(T entity) {
        getEntityManager().merge(entity);
    }
    public void remove(T entity) {
        getEntityManager().remove(getEntityManager().merge(entity));
    }
    public T find(Object id) {
        return getEntityManager().find(entityClass,(Long) id);
    }
    public List<T> findAll() {
        javax.persistence.criteria.CriteriaQuery cq =
                getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        return getEntityManager().createQuery(cq).getResultList();
    }
    public List<T> findRange(int[] range) {
        javax.persistence.criteria.CriteriaQuery cq =
                getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        q.setMaxResults(range[1] - range[0] + 1);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }
    public int count() {
        javax.persistence.criteria.CriteriaQuery cq =
                getEntityManager().getCriteriaBuilder().createQuery();
        javax.persistence.criteria.Root<T> rt = cq.from(entityClass);
        cq.select(getEntityManager().getCriteriaBuilder().count(rt));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }
}
```

Esta clase "Data_AbstractFacade" implementa todos los métodos comunes para todas las entidades. Para terminar con el ejemplo mostramos la interfaz (contrato) "Data_PostFacadeLocal" y su implementación "Data_PostFacade".

- Interfaz "Data_PostFacadeLocal.java":

```
package datasessionbeans;
import data.Post;
import java.util.List;
```

```
import javax.ejb.Local;
@Local
public interface Data_PostFacadeLocal {
   void create(Post post);
   void edit(Post post);
   void remove(Post post);
   Post find(Object id);
   List<Post> findAll();
   List<Post> findRange(int[] range);
   int count();
}
```

- *Bean* de sesión "Data_PostFacade.java":

```
package datasessionbeans;
import data.Post;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Stateless
public class Data_PostFacade extends Data_AbstractFacade<Post> implements Data_PostFa-
cadeLocal {
   @PersistenceContext(unitName = "Blog-ejbPU")
   private EntityManager em;
   @Override
   protected EntityManager getEntityManager() {
      return em;
   }
   public Data_PostFacade() {
      super(Post.class);
   }
}
```

Veamos de una forma simple cómo podemos empezar a probar nuestro código. Podemos empezar a hacer uso de los *servlets*. Crearemos un *servlet* que hará uso del EJB cuya interfaz es local "Data_UsuarioFacade". Para hacer uso de un EJB en un *servlet* (o en cualquier otro lugar), en la definición del atributo hemos de indicar la anotación "@EJB".

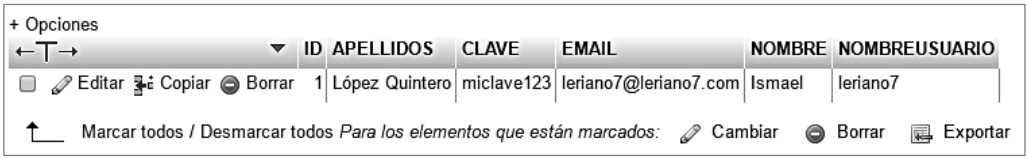
```
package servlets;
import data.Usuario;
import datasessionbeans.Data_UsuarioFacadeLocal;
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
```

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class BlogServlet extends HttpServlet {
   @EJB
   private Data_UsuarioFacadeLocal usuarioFacade;
   private Usuario usuario;
   protected void processRequest(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      usuario = new Usuario();
      usuario.setNombre("Ismael");
      usuario.setApellidos("López Quintero");
      usuario.setNombreUsuario("leriano7");
      usuario.setClave("miclave123");
      usuario.setEmail("leriano7@leriano7.com");
      try {
         usuarioFacade.create(usuario);
      } catch(Exception e) {
         System.out.println(e.getMessage());
      }
      response.setContentType("text/html;charset=UTF-8");
      try (PrintWriter out = response.getWriter()) {
         /* TODO output your page here. You may use following sample code. */
         out.println("<!DOCTYPE html>");
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet BlogServlet</title>");
         out.println("</head>");
         out.println("<body>");
         out.println("<h1>Servlet BlogServlet at " + request.getContextPath() + "</h1>");
         out.println("</body>");
         out.println("</html>");
      }
   }

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

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

```
    @Override
    public String getServletInfo() {
        return "Información del servlet";
    }
}
```

Compilamos el proyecto completo y lo desplegamos sobre el servidor de aplicaciones. Ejecutamos el *servlet*.

Y vemos lo que ha ocurrido en la base de datos.



5.4.3.2 Ejercicio 3

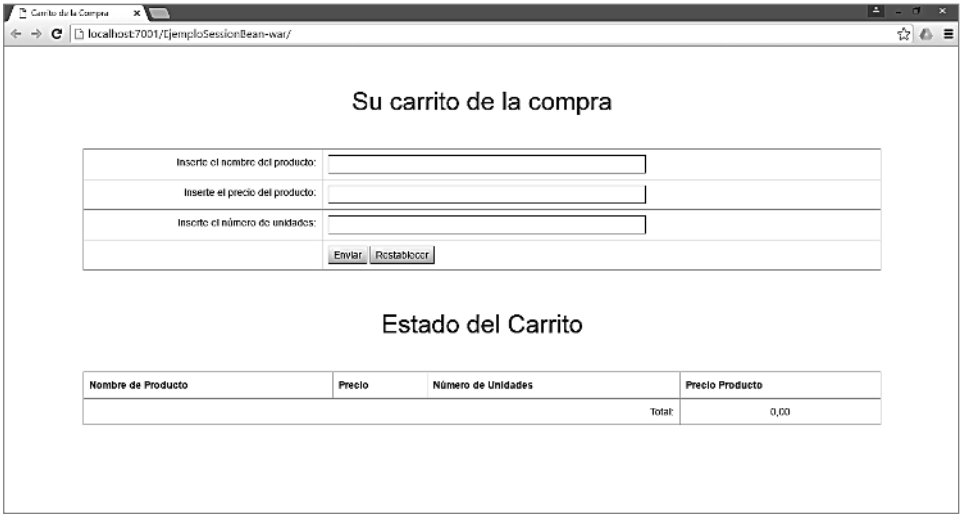
Habiéndose implementado en el texto las fachadas de acceso a datos mediante *stateless session bean* de las entidades de "Post" y "Usuario", se propone al lector la implementación de las fachadas de acceso a datos de "ComentaPost", "Chat" y "ChatActual".

5.4.3.3 Con estado: *stateful*

Estudiemos el caso de un *bean* de sesión con estado. El ejemplo más típico de uso de un *bean* con estado es el del carrito de la compra en un comercio electrónico. Pero gran cantidad de desarrolladores, la primera vez que se enfrentan a tratar este caso de uso con los *stateful session beans*, cometen el error de esperar que se creen nuevos EJB con cada sesión de usuario HTTP que recibe el *servlet*. Este razonamiento es falso, ya que los *beans stateful* crean una nueva instancia por cada cliente remoto y disponible para él sólo durante el tiempo de vida de éste. Si implementamos una aplicación web en la que el usuario haga uso de un *servlet*, no tendremos varios *stateful session beans*, sino uno sólo, propiedad del *servlet*. Por ello mismo es importante no delegar la lógica de "un carrito por cada usuario" al EJB, sino hacer el discernimiento a nivel

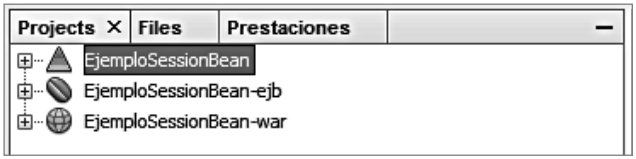
de sesión HTTP. Como el *servlet* siempre tendrá la misma instancia del EJB, podemos usar el EJB para mantener el estado de los diversos carritos.

Creemos una aplicación web en la que cada cliente que se conecte al servidor vea la siguiente pantalla:

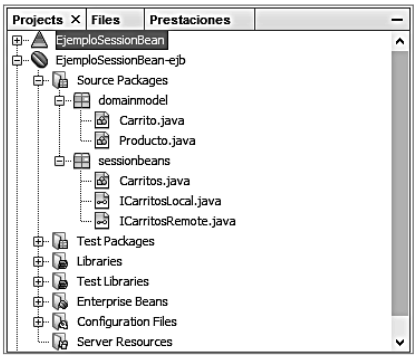


En dicho ejemplo tendremos un formulario en el que los campos (de texto libre) alimentarán a una tabla. Cada usuario que se conecte mediante HTTP a la aplicación tendrá su propio carrito y, por ende, su propia tabla. Como hemos dicho, este discernimiento debemos hacerlo mediante la sesión HTTP. La solución que se nos ocurre para este problema con los EJB *stateful* es guardar (en el *stateful* del *servlet*) un atributo de tipo "Map" que conserve el "id" de la sesión de cada usuario unido a la información de su carrito de la compra. El lector se estará preguntando si estamos usando el *stateful* EJB como si fuera un *singleton*. En efecto, lo estamos haciendo, porque el valor del *stateful* no lo encontramos en las llamadas a *servlets* a través de HTTP, sino a través de RMI, con llamadas remotas (mediante la interfaz remota) del EJB. Vayamos por partes y veamos cómo implementar el ejemplo del carrito con EJB *stateful*. La estructura de carpetas del proyecto es la siguiente:

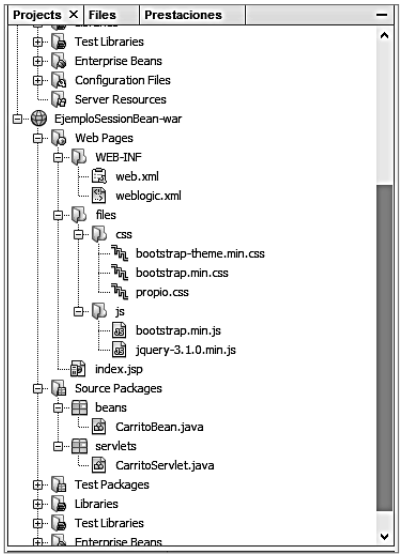
Visión general:



Módulo EJB:



Módulo web (WAR):



Para definir la vista vemos que hemos definido un JSP ("index.jsp"). Aparte, hemos importado la librería de JavaScript jQuery y hemos importado las librerías de CSS y de JavaScript del *framework* Bootstrap (*framework* de la capa de presentación). Todo ello desde sus respectivos sitios web. Son de libre uso. También podemos ver un fichero "propio.css". Con dicho fichero le damos el toque personal a nuestra vista en aquello que lo necesitemos.

Para poder trabajar las vistas a nivel de sesión, hemos definido un *bean* de sesión (no confundir con los EJB *session beans*), es decir, una clase con atributos privados, *getters & setters*, constructor sin atributos y que implementa la interfaz serializable

(como se explicó anteriormente). Veamos primero cómo podemos definir la vista que se ha mostrado a modo de prototipo:

- Clase "CarritoBean.java" (un JavaBean de sesión):

```
package beans;
import domainmodel.Producto;
import java.io.Serializable;
import java.util.List;
public class CarritoBean implements Serializable {
   public CarritoBean() {}
   private List<Producto> productos;
   private double precioTotal;
   public List<Producto> getProductos() {
      return productos;
   }
   public void setProductos(List<Producto> productos) {
      this.productos = productos;
   }
   public double getPrecioTotal() {
      return precioTotal;
   }
   public void setPrecioTotal(double precioTotal) {
      this.precioTotal = precioTotal;
   }
}
```

- Fichero CSS "propio.css":

```
div.container {
   margin-top: 1cm;
   text-align: center;
}
div.container table {
   margin: 0 auto;
   text-align: center;
}
div.container table.form th:first-child,
div.container table.form td:first-child {
   width: 30%;
   text-align: right;
}
div.container table.form td.input-field {
   text-align: left;
}
div.container table td.total {
   text-align: right;
}
```

- Página "index.jsp":

```
<%@page contentType="text/html" pageEncoding="UTF-8" session="true"
  import=
  "javax.ejb.*,javax.naming.*,sessionbeans.*,java.text.DecimalFormat,
  java.util.*,beans.*,domainmodel.*" %>
<jsp:useBean id="carritoBean" class="beans.CarritoBean" scope="session" />
<%!
  DecimalFormat decimales;
%>
<%
  decimales = new DecimalFormat("0.00");
%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <link rel="stylesheet" type="text/css"
        href="files/css/bootstrap.min.css" />
    <link rel="stylesheet" type="text/css"
        href="files/css/bootstrap-theme.min.css" />
    <link rel="stylesheet" type="text/css"
        href="files/css/propio.css" />
    <script type="text/javascript"
      src="files/js/jquery-3.1.0.min.js"></script>
    <script type="text/javascript"
      src="files/js/bootstrap.min.js"></script>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Carrito de la Compra</title>
  </head>
  <body>
    <div class="container">
      <h1>Su carrito de la compra</h1>
    </div>
    <div class="container">
      <form action="CarritoServlet" method="post">
        <table class="table table-bordered form">
          <tbody>
            <tr>
              <td>Inserte el nombre del producto: </td>
              <td class="input-field"><input type="text"
                name="nombre" size="60" maxlength="100" /></td>
            </tr>
            <tr>
              <td>Inserte el precio del producto: </td>
```

```
                <td class="input-field"><input type="text"
                    name="precio" size="60" maxlength="10" /></td>
                </tr>
                <tr>
                    <td>Inserte el número de unidades: </td>
                    <td class="input-field"><input type="text"
                        name="nunid" size="60" maxlength="10" /></td>
                </tr>
                <tr>
                    <td> </td>
                    <td class="input-field"> <input type="submit" />
                        <input type="reset" /></td>
                </tr>
            </tbody>
        </table>
    </form>
</div>
<div class="container">
    <h1>Estado del Carrito</h1>
</div>
<div class="container">
    <table class="table table-bordered">
        <thead>
            <tr>
                <th>Nombre de Producto</th>
                <th>Precio</th>
                <th>Número de Unidades</th>
                <th>Precio Producto</th>
            </tr>
        </thead>
        <tbody>
            <%
                if ((carritoBean != null) &&
                (carritoBean.getProductos() != null)) {
                    List<Producto> productos =
                            carritoBean.getProductos();
                    for (Producto p : productos) {
                        double precioTotal =
                            p.getPrecio() * (double)p.getnUnidades();
            %>
                        <tr>
                            <td>
                                <%=p.getNombre()%>
                            </td>
                            <td>
                                <%=decimales.format(p.getPrecio())%>
```

```
                    </td>
                    <td>
                      <%=p.getnUnidades()%>
                    </td>
                    <td>
                      <%=decimales.format(precioTotal)%>
                    </td>
                  </tr>
            <%
                }
              }
            %>
            <tr>
              <td colspan="3" class="total">Total: </td>
              <%
                if ((carritoBean != null) &&
                    (carritoBean.getProductos() != null)) {
              %>
                    <td>
                      <%=decimales.format(carritoBean.getPrecioTotal())%>
                    </td>
              <%
                } else {
              %>
                    <td><%=decimales.format(0.0)%></td>
              <%
                }
              %>
            </tr>
          </tbody>
        </table>
      </div>
    </body>
</html>
```

Antes de estudiar cómo trata el *servlet* la información de la sesión, veamos cómo implementamos el módulo EJB.

Las clases del modelo del dominio están en el paquete que se llama *domainmodel*. Representan la información de cada uno de los carritos con los que trabajará el EJB *stateful*.

- Clase "Producto.java":

```
package domainmodel;
import java.io.Serializable;
public class Producto implements Serializable{
```

```
    private String nombre;
    private double precio;
    private int nUnidades;
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public double getPrecio() {
        return precio;
    }
    public void setPrecio(double precio) {
        this.precio = precio;
    }
    public int getnUnidades() {
        return nUnidades;
    }
    public void setnUnidades(int nUnidades) {
        this.nUnidades = nUnidades;
    }
}
```

- Clase "Carrito.java":

```
package domainmodel;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
public class Carrito implements Serializable {
    private List<Producto> productos;
    private double valorTotal;
    public Carrito() {
        this.productos = new ArrayList<Producto>();
    }
    public List<Producto> getProductos() {
        return productos;
    }
    public void setProductos(List<Producto> productos) {
        this.productos = productos;
    }
    public double getValorTotal() {
        return valorTotal;
    }
    public void setValorTotal(double valorTotal) {
        this.valorTotal = valorTotal;
    }
    public void anadirProductoCarrito(Producto p) {
```

```
        if ((p != null) && (this.productos != null)) {
            boolean encontrado = false;
            int i = 0;
            for (Producto producto : this.productos) {
                if (producto.getNombre().equals(p.getNombre())) {
                    encontrado = true;
                    break;
                } else {
                    i++;
                }
            }
            if (encontrado) {
                Producto esteProducto = this.productos.get(i);
                esteProducto.setnUnidades(esteProducto.getnUnidades()
                        + p.getnUnidades());
            } else {
                this.productos.add(p);
            }
        }
    }
    public void eliminarProductoCarrito(Producto p) {
        if ((p != null) && (this.productos != null)) {
            boolean encontrado = false;
            int i = 0;
            for (Producto producto : this.productos) {
                if (producto.getNombre().equals(p.getNombre())) {
                    encontrado = true;
                    break;
                } else {
                    i++;
                }
            }
            if (encontrado) {
                this.productos.remove(i);
            }
        }
    }
    public double getPrecioTotalCarrito() {
        if (this.productos != null) {
            double total = 0.0;
            for (Producto p : this.productos) {
                total = total + (p.getPrecio() * p.getnUnidades());
            }
            return total;
        } else {
            return 0.0;
        }
    }
}
```

Estudiemos ahora el componente EJB *stateful* "Carritos". Indicar que se implementa una interfaz local y una interfaz remota.

- Interfaz "ICarritosLocal.java":

```
package sessionbeans;
import domainmodel.Producto;
import java.util.List;
import javax.ejb.Local;
@Local
public interface ICarritosLocal {
   public void usarCarrito(String id); // Identificador de la sesión.
   public void anadirProductoCarrito(String id, Producto p);
   public void eliminarProductoCarrito(String id, Producto p);
   public double getPrecioTotalCarrito(String id);
   public List<Producto> getProductos(String id);
   public void remove(String id);
   public void remove(); // Remove de Stateful Session Bean.
}
```

- Interfaz "ICarritosRemote.java":

```
package sessionbeans;
import domainmodel.Producto;
import java.util.List;
import javax.ejb.Remote;
@Remote
public interface ICarritosRemote {
   public void usarCarrito(String id); // Identificador de la sesión.
   public void anadirProductoCarrito(String id, Producto p);
   public void eliminarProductoCarrito(String id, Producto p);
   public double getPrecioTotalCarrito(String id);
   public List<Producto> getProductos(String id);
   public void remove(String id);
   public void remove(); // Remove de Stateful Session Bean.
}
```

- EJB *stateful* "Carritos.java". Es el componente que implementa el mapa de datos:

```
package sessionbeans;
import domainmodel.Carrito;
import domainmodel.Producto;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;
```

```
@Stateful
public class Carritos implements ICarritosLocal, ICarritosRemote {
   private HashMap<String,Carrito> carritos;
   public Carritos() {
      this.carritos = new HashMap<>();
   }
   @Override
   public void usarCarrito(String id) {
      Carrito carrito = this.carritos.get(id);
      if (carrito == null) {
         carrito = new Carrito();
         this.carritos.put(id, carrito);
      }
   }
   @Override
   public List<Producto> getProductos(String id) {
      Carrito carrito = this.carritos.get(id);
      if (carrito == null) {
         return new ArrayList<Producto>();
      } else {
         return carrito.getProductos();
      }
   }
   @Override
   public void anadirProductoCarrito(String id, Producto p) {
      Carrito carrito = this.carritos.get(id);
      if (carrito != null) {
         carrito.anadirProductoCarrito(p);
      }
   }
   @Override
   public void eliminarProductoCarrito(String id, Producto p) {
      Carrito carrito = this.carritos.get(id);
      if (carrito != null) {
         carrito.eliminarProductoCarrito(p);
      }
   }
   @Override
   public double getPrecioTotalCarrito(String id) {
      Carrito carrito = this.carritos.get(id);
      if (carrito != null) {
         return carrito.getPrecioTotalCarrito();
      } else {
         return 0.0;
      }
   }
```

```
    @Override
    @Remove
    public void remove() {
        this.carritos = null;
    }
    @Override
    public void remove(String id) {
        Carrito carrito = this.carritos.get(id);
        if (carrito != null) {
            this.carritos.remove(id);
        }
    }
}
```

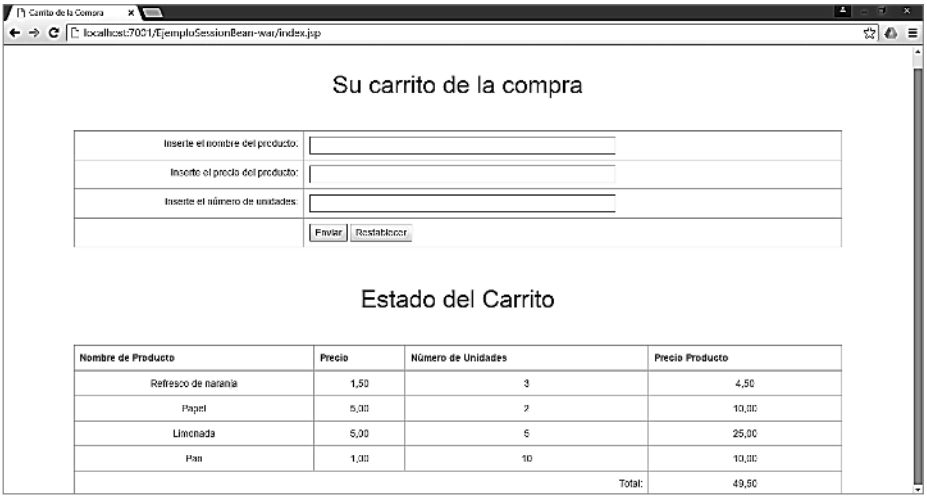
Finalmente, volvemos a la vista (proyecto WAR) para ver cómo implementamos el *servlet*:

- "CarritoServlet.java":

```
package servlets;
import beans.CarritoBean;
import domainmodel.Producto;
import java.io.IOException;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import sessionbeans.ICarritosLocal;
public class CarritoServlet extends HttpServlet {
    @EJB
    private ICarritosLocal carritos;
    private CarritoBean carritoBean;
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {
        synchronized (this) {
            HttpSession session = request.getSession();
            String idSesion = session.getId();
            carritos.usarCarrito(idSesion);
            if (session.getAttribute("carritoBean") == null) {
                carritoBean = new CarritoBean();
                session.setAttribute("carritoBean", carritoBean);
            } else {
                carritoBean = (CarritoBean) session.getAttribute("carritoBean");
            }
```

```
        String nombreProducto = request.getParameter("nombre");
        String precioProducto = request.getParameter("precio");
        String nUnidadesProducto = request.getParameter("nunid");
        double precio = 0.0;
        int nUnidades = 0;
        try {
           precio = Double.parseDouble(precioProducto);
           nUnidades = Integer.parseInt(nUnidadesProducto);
        } catch (Exception e) {}
        if ((precio != 0.0) && (nUnidades != 0)) {
           // Los datos se han introducido bien, modificamos el EJB stateful.
           Producto p = new Producto();
           nombreProducto = new String(nombreProducto.getBytes("ISO-8859-1"), "UTF-8");
           p.setNombre(nombreProducto);
           p.setPrecio(precio);
           p.setnUnidades(nUnidades);
           carritos.anadirProductoCarrito(idSesion,p);
           System.out.println("El carrito tiene " +
                 carritos.getProductos(idSesion).size() + " elementos".);
        }
        carritoBean.setProductos(carritos.getProductos(idSesion));
        carritoBean.setPrecioTotal(carritos.getPrecioTotalCarrito(idSesion));
        response.sendRedirect("index.jsp");
     }
  }
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
     processRequest(request, response);
  }
  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
     processRequest(request, response);
  }
  @Override
  public String getServletInfo() {
     return "Descripción del servlet CarritoServlet";
  }
}
```

Vemos cómo llamamos al *stateful session bean* en la llamada a "usarCarrito". Se añadirá un carrito al mapa por cada "id" de sesión. Ésta es la solución propuesta para la aplicación web del carrito de la compra con *stateful session beans* (usándolo como si fuera un *singleton*). Ya que lo llamamos siempre desde el *servlet*, éste tendrá inyectada la misma instancia del EJB (siempre será el mismo cliente). Veamos el funcionamiento de la aplicación desarrollada.



Los EJB *stateful* muestran su potencial en la llamada haciendo uso de la interfaz remota, a través de RMI. El estudio de RMI debe realizarse en tres estadios diferentes, para entenderlo mejor:

- Implemantación de RMI en la JVM.
- Implementación de RMI en el servidor de aplicaciones.
- Implementación de RMI mediante una interfaz de EJB remota.

5.4.3.3.1 RMI en la JVM local

RMI consiste, como su nombre bien indica, en la invocación remota de métodos. Dicho estándar conforma una librería que es parte de la JSE. La tecnología es heredera de RPC *(Remote Procedure Call)*, versión de RMI para la programación estructurada. Lo que pretende es asociar al registro local de la máquina un método en un punto de red, que será visible por los demás equipos conectados a la misma red (en nuestro caso la red local).

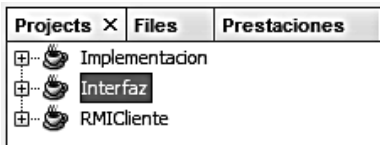
Para implementar RMI debemos exponer una interfaz remota en una librería separada de la implementación de la interfaz. Evidentemente, para poder implementar dicha interfaz debemos importar en el proyecto de implementación el archivo ".jar" de la librería. La idea es la siguiente:

- Implementamos en nuestra máquina una interfaz remota que será pública.
- Creamos una aplicación Java que añadirá dicha implementación al registro.

- Facilitamos a los clientes que quieran hacer uso de nuestra implementación el nombre de acceso al registro de nuestra dirección IP.
- Los clientes se conectarán a nuestra máquina haciendo uso de la interfaz remota y del nombre de acceso al registro que les habremos facilitado.

La idea subyacente es aislar el uso de los métodos de la implementación de éstos, creando sistemas débilmente acoplados. El lector se estará preguntando si llevamos una copia del objeto a cada cliente. La respuesta es sí y a la vez no. Lo que se traslada al cliente es una imagen espejo de nuestro objeto (una imagen sincronizada), ya que los cambios en la "copia" afectan directamente al original. Lo que el cliente obtiene es un *proxy*. Para implementar RMI en versiones anteriores de JSE debíamos compilar la clase que accedía al registro, llamada *skeleton*, creando una clase *stub*. Esta forma de implementar RMI está obsoleta y no es necesario llamar a *rmic* desde la línea de comandos.

Veamos cómo implementar RMI con un sencillo ejemplo. El ejemplo que veremos será el archiconocido "Hola Mundo" pero implementado mediante RMI. Como bien hemos observado, RMI en sí es una característica de JSE. Por lo tanto, para programar este ejemplo podemos apagar el servidor de aplicaciones. Todo lo que necesitamos es dos aplicaciones Java (una de implementación y otra cliente), así como una librería que expone la interfaz remota. O sea, tendremos tres proyectos Java.



En el proyecto "Interfaz" colocamos la interfaz remota. Dicho proyecto es una librería de clases. Los otros dos proyectos son aplicaciones Java JSE. El proyecto "Implementacion" es la implementación de la interfaz y el proyecto "RMICliente" es el programa que se ejecutará en cualquier equipo de nuestra red.

Veamos el código de cada uno de los proyectos:

- Código de la interfaz remota en la librería Interfaz:

```
package ejemplo;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Saludo extends Remote {
   public String saludar() throws RemoteException;
}
```

- Código de la implementación de la interfaz. Debemos importar el fichero "Interfaz.jar":

```
package ejemplo;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class Implementacion extends UnicastRemoteObject implements Saludo {
  protected Implementacion() throws RemoteException {
    super();
  }

  @Override
  public String saludar() throws RemoteException {
    return "Hola Mundo";
  }
}
```

Y del mismo modo vemos el programa principal que accede al registro, situando en él una instancia de la clase "Implementacion":

```
package ejemplo;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class PrincipalRMI {
  public static void main(String[] args) {
    try {
      Saludo saludo = new Implementacion();
      Registry registro = LocateRegistry.createRegistry(9091);
      registro.rebind("saludo", saludo);
      System.out.println("Añadimos al registro");
    } catch (Exception ex) {
      System.out.println(ex.getMessage());
    }
  }
}
```

De este modo, estamos situando la implementación de la interfaz remota en el puerto de red 9091 de la máquina local. Aquellos clientes que quieran hacer uso de la implementación tendrán:

- Acceso a la librería que define la interfaz remota.
- Nuestra dirección IP.
- El puerto en el que hemos situado la implementación (9091).
- El nombre que le hemos asignado en el registro de nuestra máquina ("saludo").

La dirección IP de nuestra máquina es 192.168.0.101. Facilitando dicha información y el puerto pueden escribir la aplicación cliente.

- La aplicación "RMICliente":

```
package rmicliente;
import ejemplo.Saludo;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class RMICliente {
   public static void main(String[] args) {
      try {
         Registry registro = LocateRegistry.getRegistry("192.168.0.101", 9091);
         Saludo saludo = (Saludo) registro.lookup("saludo");
         System.out.println(saludo.saludar());
      } catch (Exception ex) {
         System.out.println(ex.getMessage());
      }
   }
}
```

Ejecutando la aplicación cliente tenemos la salida esperada:

Hola Mundo

5.4.3.3.2 RMI en el servidor de aplicaciones

Una vez entendido el anterior ejemplo, nos disponemos a hacer lo mismo en Oracle WebLogic Server. Damos el paso al servidor de aplicaciones pero aún no usamos EJB. La idea es dejar la aplicación de implementación corriendo en el servidor. Para ello hemos de setear dicha clase como una clase de arranque del servidor. Los clientes se conectarán ahora a la implementación que residirá en el servidor de aplicaciones. Seguimos teniendo tres proyectos JSE.

- "Interfaz": interfaz remota.
- "Implem": proyecto a desplegar en el servidor de aplicaciones que incluye la implementación.
- "Cliente": programa que ejecutaremos desde cualquier máquina conectada a la red del servidor de aplicaciones.

Para conectarnos como clientes RMI a WebLogic usamos el protocolo T3. Cada servidor de aplicaciones (JBoss, GlassFish, JOnAs...) tendrá su configuración y protocolo para conectarse al servidor de aplicaciones. En nuestro caso, identificamos el registro como el contexto del servidor. La configuración para setear y leer información del contexto es la siguiente:

```
String url = "t3://" + hostname + ":" + port;
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
env.put(Context.PROVIDER_URL, url);
env.put("weblogic.jndi.connectTimeout", new Long(15000));
env.put("weblogic.jndi.responseReadTimeout", new Long(15000));
env.put(Context.SECURITY_PRINCIPAL, "admin");
env.put(Context.SECURITY_CREDENTIALS, "admin_password");
new InitialContext(env);
```

Ésta es una nota técnica para saber cómo acceder al contexto en WebLogic. Este código debe ajustarlo el programador para su servidor elegido.

El código de la interfaz, situado en una librería, ya lo conocemos:

```
package ejemplo;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Saludo extends Remote {
  public String saludar() throws RemoteException;
}
```

- Código de la implementación. Es una aplicación JSE con un método *main*. Hemos situado en una sola clase la implementación de la interfaz y el método *main*.

```
package implem;
import ejemplo.Saludo;
import java.rmi.RemoteException;
import java.util.Hashtable;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
public class Implem implements Saludo {
  private static String hostname = "localhost";
  private static int port = 7001;
  private static String url = "";
  private static Context ctx = null;
  private static Saludo saludo;
```

```
    @Override
    public String saludar() throws RemoteException {
        return "Hola mundo";
    }
    public static void main(String[] args) {
        try {
            ctx = getInitialContext();
            saludo = new Implem();
            ctx.rebind("saludo", saludo);
        } catch (Exception ex) {
            Logger.getLogger(Implem.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
    private static Context getInitialContext() throws NamingException {
        url = "t3://" + hostname + ":" + port;
        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "weblogic.jndi.WLInitialContextFactory");
        env.put(Context.PROVIDER_URL, url);
        env.put("weblogic.jndi.connectTimeout", new Long(15000));
        env.put("weblogic.jndi.responseReadTimeout", new Long(15000));
        env.put(Context.SECURITY_PRINCIPAL, "admin");
        env.put(Context.SECURITY_CREDENTIALS, "admin_password");
        return new InitialContext(env);
    }
}
```

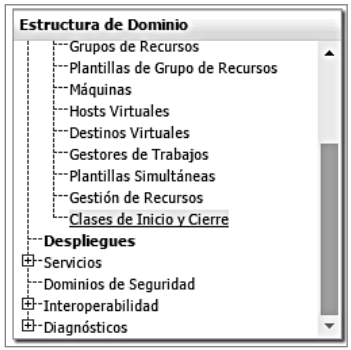
La idea es compilar esta clase y colocarla como una clase de arranque de WebLogic. Veamos cómo hacerlo:

- Debemos crear un directorio en la ruta de nuestro dominio, a saber: "C:\Oracle\Middleware\Oracle_Home\user_projects\domains\learn\" para colocar el fichero ".jar" ejecutable que hemos generado al compilar el proyecto "Implem". Dicha carpeta la llamamos, por ejemplo, *jar*. La ruta sería: "C:\Oracle\Middleware\Oracle_Home\user_projects\domains\learn\jar".
- Copiamos en dicho fichero el ".jar": "Implem.jar".
- Ajustamos el *classpath* del servidor. Para ello acudimos al fichero: "C:\Oracle\Middleware\Oracle_Home\user_projects\domains\learn\bin\startWebLogic.cmd". Justo al final del fichero, antes de la sección *echo* (salida por consola), insertamos la siguiente línea:

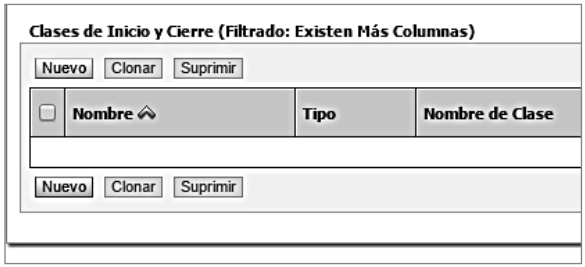
  "CLASSPATH = %CLASSPATH%;%DOMAIN_HOME%\jar\Implem.jar".

Creamos la clase de arranque. La localización de la clase será "implem.Implem". Veamos algunas capturas de pantalla.

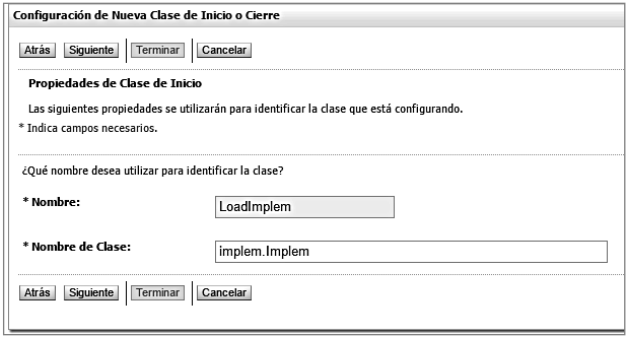
En la estructura del dominio "learn", seleccionamos la opción "Clases de Inicio y Cierre".



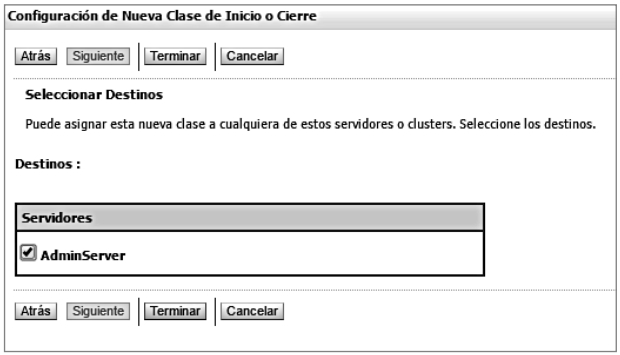
Creamos nueva clase.



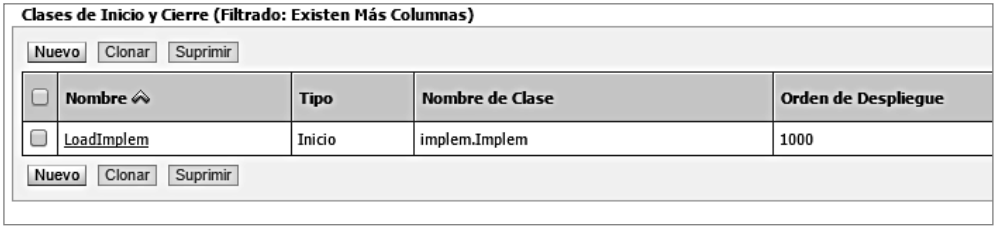
Indicamos el nombre que será tomado del *classpath*.



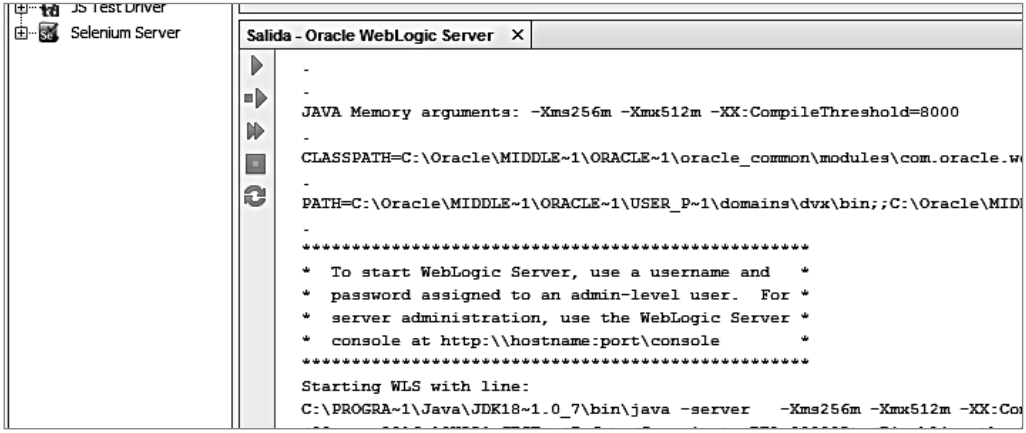
La desplegamos en "AdminServer" del dominio.



Vemos la clase de inicio instalada.



Reiniciamos el servidor y vemos la variable *classpath*.



Y finalmente veremos el servicio RMI del servidor colgando del árbol JNDI con el nombre "saludo".

Finalmente desarrollamos la aplicación cliente con el siguiente código:

```
package cliente;
import ejemplo.Saludo;
import java.util.Hashtable;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
public class Cliente {
   private static String hostname = "192.168.0.101";
   private static int port = 7001;
   private static String url = "";
   private static Context ctx = null;
   public static void main(String[] args) {
      try {
         ctx = getInitialContext();
         Saludo saludo = (Saludo) ctx.lookup("saludo");
         System.out.println(saludo.saludar());
      } catch (Exception ex) {
         Logger.getLogger(Cliente.class.getName()).log(Level.SEVERE, null, ex);
      }

   }
   private static Context getInitialContext() throws NamingException {
      url = "t3://" + hostname + ":" + port;
      Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
      env.put(Context.PROVIDER_URL, url);
      return new InitialContext(env);
   }
}
```

Hola Mundo

5.4.3.3.3 RMI. Interfaz remota de *stateful* EJB

Hemos tenido una aproximación para ver cómo podemos implementar RMI en WebLogic. Ahora veremos cómo acceder a un *stateful* EJB mediante interfaz remota, usando precisamente RMI. De esta forma podremos entender en qué consisten las sesiones (diferentes a las sesiones HTTP). Ahora sí podemos hablar de *beans* de sesión que conservan estado. Este estado es conservado durante la vida del cliente. Si recordamos el carrito de la compra que implementamos con un EJB *stateful*, tendríamos un carrito de la compra por cada cliente que se conecta al EJB mediante RMI.

Los *stateful session beans* pueden pasar a estado "pasivo" por tiempo de inactividad del cliente. El estado pasivo indica que la imagen del *bean* pasa de memoria principal a memoria secundaria (disco) del servidor. Por ello mismo, el objeto es sometido a un proceso de *marshalling* o serialización. Todos los atributos del EJB deben ser tipos primitivos o implementar la interfaz serializable. El *bean* vuelve a memoria principal cuando el cliente vuelve a invocarlo. Si un cliente termina su ejecución, el EJB *stateful* es destruido.

Para no repetir el código del carrito, en gran parte igual, vamos a implementar otro ejemplo. Cada *bean* de sesión guarda un entero. Este entero se incrementa en uno cada vez que se consulta. Evidentemente, cada cliente tendrá su contador.

Crearemos tres proyectos:

- "InterfazRemota": define la interfaz remota del EJB *stateful*.
- "MiSessionBean-ejb": módulo contenedor en el que definiremos nuestro EJB.
- "ClienteEJBRemoto": Aplicación JSE que ocupará el rol de cliente.

Veamos cómo queda el código de los tres proyectos:

- "InterfazRemota". Interfaz "EjemploSessionBeanRemote.class":

```
package ejemplo;
import javax.ejb.Remote;
@Remote
public interface EjemploSessionBeanRemote {
  public int getContador();
}
```

- EJB "EjemploSessionBean":

```
package ejemplo;
import javax.ejb.Stateful;
@Stateful
public class EjemploSessionBean implements EjemploSessionBeanRemote {
  private int contador;
  public EjemploSessionBean() {
```

```
      contador = 0;
   }
   @Override
   public int getContador() {
      contador++;
      return contador;
   }
}
```

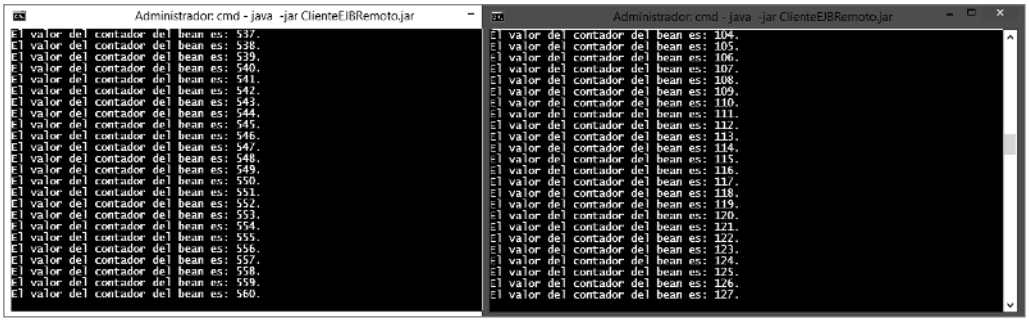
Para poder implementar el cliente debemos importar la librería "wlthint3client.jar" ubicada en "C:\Oracle\Middleware\Oracle_Home\wlserver\server\lib". Por supuesto, también debemos importar la librería de la interfaz remota.

- "ClienteEJBRemoto.class":

```
package clienteejbremoto;
import ejemplo.EjemploSessionBeanRemote;
import java.util.Hashtable;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
public class ClienteEJBRemoto {
   private static String hostname = "192.168.0.101";
   private static int port = 7001;
   private static Context ctx = null;
   public static void main(String[] args) {
      try {
         ctx = getInitialContext();
         EjemploSessionBeanRemote sbr = (EjemploSessionBeanRemote)
         ctx.lookup("java:global.MiSessionBean.MiSessionBean-ejb.EjemploSessionBean"
               + "!ejemplo.EjemploSessionBeanRemote");
         int contador = -1;
         while(contador < 1000) {
            contador = sbr.getContador();
            Thread.sleep(100);
            System.out.println("El valor del contador del bean es: "+contador+"".);
         }
      } catch (Exception ex) {
         System.out.println(ex.getMessage());
      }
   }
   private static Context getInitialContext() throws NamingException {
      Context contexto = null;
      String url = "t3://" + hostname + ":" + port;
      Hashtable env = new Hashtable();
```

```
        env.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
        env.put(Context.PROVIDER_URL, url);
        contexto = new InitialContext(env);
        return contexto;
    }
}
```

La salida del programa desde dos clientes diferentes es la siguiente:



Evidentemente, cada cliente conserva su estado del *bean*, porque tiene una sesión asociada y un EJB *stateful* propio.

5.4.3.4 Instancia única: *singleton*

El desarrollo de *software* se hizo mucho más escalable y mantenible gracias al paradigma de la programación orientada a objetos. Como se ha explicado a lo largo de este manual, un objeto es una instancia de una clase. Pero junto con la programación orientada a objetos aparecieron un conjunto de patrones de diseño de *software*. Si queremos estudiar los patrones de diseño para que la arquitectura de nuestra aplicación siga las soluciones más efectivas que han planteado los desarrolladores a lo largo de los años, tenemos un compendio de ellos en el libro *Patrones de diseño: elementos de software orientado a objetos reutilizables* (autores: Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides). Al grupo de estos cuatro arquitectos de *software* se les conoce como *La banda de los cuatro (The Gang of Four* o *GoF)*. Cuando en textos de *software* leamos que el diseño de la arquitectura sigue los principios de GoF, sabremos que se siguen los patrones de diseño indicados en dicho libro.

Un patrón de diseño es aquel en el que queremos una instancia única de una clase. El patrón *singleton* se encarga de este cometido. No queremos que los programadores puedan crear tantas instancias de la clase como quieran haciendo uso de la instrucción *new*. Sólo debe haber una instancia de dicha clase en la aplicación (un solo objeto). ¿Cómo se consigue esto?:

- Mediante un constructor privado que sólo podrá usar la propia clase.
- Mediante un método estático "getInstancia" que devolverá la única instancia de la clase.

Veamos cómo implementar el patrón de diseño *singleton*. Creemos un proyecto JSE. Para probar el correcto funcionamiento del *singleton* crearemos una aplicación multi-hilo en la que accederemos a un contador como el anterior. Pero esta vez el valor del contador será el mismo para los distintos hilos de la aplicación.

Veamos el código de la clase *singleton*:

```
package singletonapp;
public class Singleton {
   private int contador;
   private static Singleton instancia = null;
   private Singleton() {
      this.contador = 0;
   }
   public static Singleton getInstancia() {
      if (instancia == null) {
         instancia = new Singleton();
      }
      return instancia;
   }
   public int getContador() {
      contador++;
      return contador;
   }
}
```

Tendremos tres hilos en la aplicación que harán uso de la clase *singleton*. Veamos el código de los hilos:

```
package singletonapp;
public class Hilo extends Thread {
   private Singleton instancia;
   private final static int MAX = 100;
   public Hilo(String nombre) {
      super(nombre);
      this.instancia = Singleton.getInstancia();
      this.start();
   }
   public void run() {
      try {
         int valor = 0;
         while (valor <= MAX) {
            System.out.println("Valor obtenido por el hilo "
                  + this.getName() + ": " + valor + "".);
            Thread.sleep(500);
            synchronized (instancia) {
               valor = instancia.getContador();
```

```
            }
          }
        } catch (Exception e) {
          System.out.println("Se ha producido un error");
        }
    }
}
```

Y ahora veamos el código del programa principal que echa a andar la aplicación:

```
package singletonapp;
public class SingletonApp {
   public static void main(String[] args) {
      Hilo hilo1 = new Hilo("primero");
      Hilo hilo2 = new Hilo("segundo");
      Hilo hilo3 = new Hilo("tercero");
      try {
         hilo1.join();
         hilo2.join();
         hilo3.join();
      } catch (Exception e) {
         System.out.println("Se produjo un error");
      }
   }
}
```

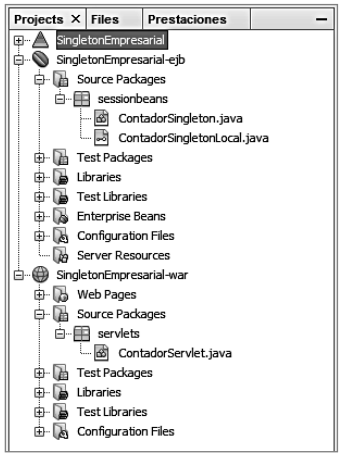
Vemos que el programa crea tres hilos. Cada uno de estos hilos, en su constructor, llama al método "getInstancia" del *singleton*. En dicha instancia estará el valor del entero que será compartido por toda la aplicación. Si en vez de llamar al método "getInstancia()" llamáramos al constructor con *new*, cada hilo tendría su propio valor del contador. Y esto es precisamente lo que no queremos. Veamos la salida del programa:

```
Valor obtenido por el hilo tercero: 0.
Valor obtenido por el hilo segundo: 0.
Valor obtenido por el hilo primero: 0.
Valor obtenido por el hilo tercero: 3.
Valor obtenido por el hilo primero: 2.
Valor obtenido por el hilo segundo: 1.
Valor obtenido por el hilo segundo: 6.
Valor obtenido por el hilo tercero: 4.
Valor obtenido por el hilo primero: 5.
Valor obtenido por el hilo tercero: 7.
Valor obtenido por el hilo segundo: 8.
Valor obtenido por el hilo primero: 9.
Valor obtenido por el hilo segundo: 10.
Valor obtenido por el hilo tercero: 11.
```

```
Valor obtenido por el hilo primero: 12.
Valor obtenido por el hilo tercero: 14.
Valor obtenido por el hilo primero: 15.
Valor obtenido por el hilo segundo: 13.
Valor obtenido por el hilo segundo: 16.
Valor obtenido por el hilo primero: 17.
Valor obtenido por el hilo tercero: 18.
Valor obtenido por el hilo tercero: 19.
Valor obtenido por el hilo primero: 21.
Valor obtenido por el hilo segundo: 20.
Valor obtenido por el hilo segundo: 22.
Valor obtenido por el hilo tercero: 23.
Valor obtenido por el hilo primero: 24.
Valor obtenido por el hilo tercero: 26.
Valor obtenido por el hilo segundo: 27.
Valor obtenido por el hilo primero: 25.
Valor obtenido por el hilo segundo: 30.
Valor obtenido por el hilo tercero: 29.
Valor obtenido por el hilo primero: 28.
Valor obtenido por el hilo primero: 32.
Valor obtenido por el hilo segundo: 33.
Valor obtenido por el hilo tercero: 31.
Valor obtenido por el hilo tercero: 34.
Valor obtenido por el hilo primero: 35.
Valor obtenido por el hilo segundo: 36.
Valor obtenido por el hilo primero: 37.
Valor obtenido por el hilo segundo: 38.
Valor obtenido por el hilo tercero: 39.
Valor obtenido por el hilo tercero: 42.
Valor obtenido por el hilo primero: 41.
Valor obtenido por el hilo segundo: 40.
Valor obtenido por el hilo segundo: 43.
Valor obtenido por el hilo primero: 45.
Valor obtenido por el hilo tercero: 44.
Valor obtenido por el hilo segundo: 48.
Valor obtenido por el hilo primero: 47.
Valor obtenido por el hilo tercero: 46.
Valor obtenido por el hilo primero: 50.
Valor obtenido por el hilo tercero: 49.
```

Vemos que existe un pequeño desfase entre el valor obtenido por el hilo y el momento en que se muestra en pantalla. Pero ésta es precisamente una característica de la programación concurrente. Cada hilo se ejecuta de forma independiente, y no secuencialmente, compartiendo únicamente la memoria de la aplicación.

El propio servidor de aplicaciones nos ofrece el patrón *singleton* implementado. Dicho patrón no es más que un EJB. El EJB se anota con "@Singleton". Imaginemos una aplicación web en la que queremos mantener a nivel de aplicación el número de peticiones HTTP realizadas al servidor (mediante el método GET o el método POST). Es un dato compartido por toda la aplicación. Por lo tanto, ¿por qué no implementarlo con un *singleton*? Creemos una nueva aplicación empresarial.



En dicho proyecto debemos prestar atención al EJB "ContadorSingleton" y al *servlet* "ContadorServlet". Veamos el *singleton* con su interfaz local:

```
package sessionbeans;
import javax.ejb.Local;
@Local
public interface ContadorSingletonLocal {
   public void incrementaContador();
   public int getContador();
}
```

Singleton implementando la interfaz:

```
package sessionbeans;
import javax.ejb.Singleton;
@Singleton
public class ContadorSingleton implements ContadorSingletonLocal {
   private int contador;
   public ContadorSingleton() {
      this.contador = 0;
   }
   @Override
   public void incrementaContador() {
      this.contador++;
```

```
    }
    @Override
    public int getContador() {
        return this.contador;
    }
}
```

Veamos el *servlet* (sin hacer uso de páginas JSP) para ver cómo usar el *singleton*. La instancia en el servidor seguirá estando disponible aunque cerremos todas las ventanas del navegador:

```
package servlets;
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import sessionbeans.ContadorSingletonLocal;
public class ContadorServlet extends HttpServlet {
    @EJB
    private ContadorSingletonLocal contadorSingleton;
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        contadorSingleton.incrementaContador();
        int contador = contadorSingleton.getContador();
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet ContadorServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet ContadorServlet at " +
                    request.getContextPath() + "</h1>");
            out.println("<p>El número de veces que se ha accedido al "
                    + "servidor es: " + contador + "</p>");
            out.println("</body>");
            out.println("</html>");
        }
```

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

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

    @Override
    public String getServletInfo() {
        return "Descripción del servlet";
    }
}
```

Finalmente veamos cómo accedemos al valor del *singleton* desde una ventana de Google Chrome y otra ventana de Mozilla Firefox:

- Google Chrome:

- Mozilla Firefox:

¿Qué pasaría si implementamos el ejemplo del carrito que vimos en el apartado de *stateful session bean* con un *singleton*? Pues ¡absolutamente nada! Porque, precisamente, al ser siempre el mismo cliente, usamos dicho *stateful* como si fuera un *singleton*. El EJB: "@Stateful".

```
public class Carritos implements ICarritosLocal, ICarritosRemote {
...
}
```

Lo redeclaramos de la siguiente forma:

```
@Singleton
public class Carritos implements ICarritosLocal, ICarritosRemote {
...
}
```

El funcionamiento de la aplicación no se altera en absoluto.

5.4.4 *Beans* orientados a mensajes

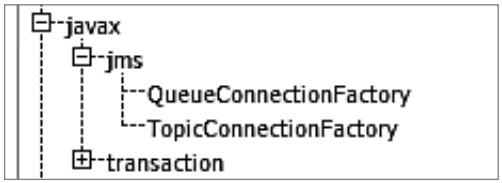
El Servicio de Mensajería de Java (*Java Messaging Service* o JMS) es la solución propuesta para las comunicaciones asíncronas en la JEE. Evidentemente, y como parte de la JEE, se nos ofrece un conjunto de clases dentro de la API para poder trabajar con dicho sistema. Los *beans* orientados a mensajes son la solución empresarial para recibir los mensajes.

La arquitectura de JMS propone dos esquemas de conexiones:

- **Conexión punto a punto.** La cual permite un canal de comunicaciones en la que sólo puede haber un emisor y un receptor. El emisor debe configurarse según los ajustes del servidor de aplicaciones. El receptor será el MDB que habremos configurado para dicho canal.
- **Conexión por tema/suscriptor.** Este esquema permite múltiples receptores para un mismo emisor, siempre que los receptores se hayan suscrito al tema para el cual el emisor envía mensajes. Tendremos varios MDB por cada tema.

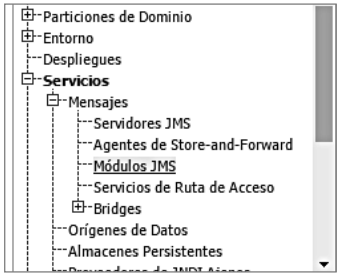
En el servidor de aplicaciones es necesaria una infraestructura concreta para poder hacer uso de JMS. En concreto necesitamos dos factorías (fábricas) de conexiones. Una fábrica para las conexiones punto a punto y otra fábrica para las conexiones por tema. De dichas factorías obtendremos las sesiones que se enlazarán con los canales, que tendremos que setear en el servidor de aplicaciones.

¿Qué viene ya seteado en Oracle WebLogic Server y qué hay que configurar? Recordemos siempre que WebLogic es el servidor de aplicaciones que se ha elegido en este manual por su robustez, pero debemos estudiar las diferentes opciones en el servidor que elijamos para nuestra aplicación. En Oracle WebLogic Server tenemos por defecto creadas dos factorías. Observemos el árbol JNDI de nuestro dominio:

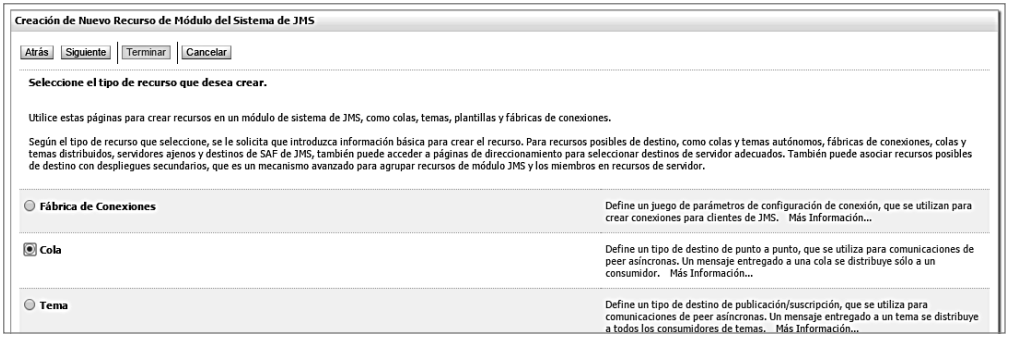


Por lo tanto, en WebLogic no será necesario crear dichas factorías. Tendremos que crear los canales para los mensajes y casarlos con las factorías de conexiones, mediante las sesiones que abriremos. Cada sesión tendrá su tipo de notificación. Normalmente en nuestro desarrollo usaremos la notificación en la que el cliente sabrá cuándo debe recibir un mensaje (cuando el mensaje exista).

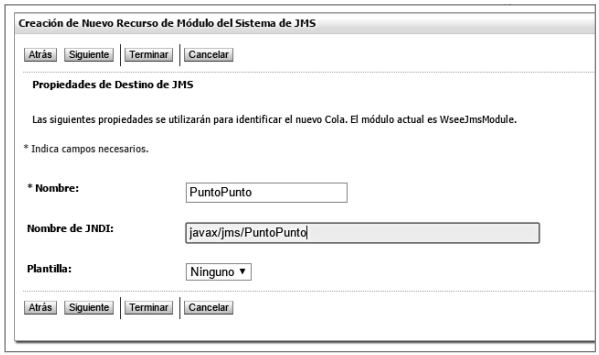
Vamos a crear los canales. Empecemos viendo un ejemplo de conexión punto a punto. Accediendo a la consola de administración de WebLogic, en concreto al panel de administración de la estructura del dominio, seleccionamos la opción "Módulos JMS".



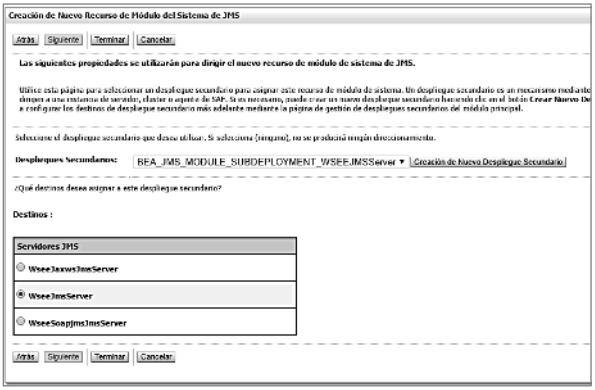
Seleccionando el módulo "WseeJmsModule" *(WebLogic Server Enterprise Edition JMS Module)*, veremos los canales disponibles. Queremos crear un canal para conexiones punto a punto (tipo cola). Seleccionamos crear un nuevo canal (tipo de recurso), y luego seleccionamos el de tipo "Cola".



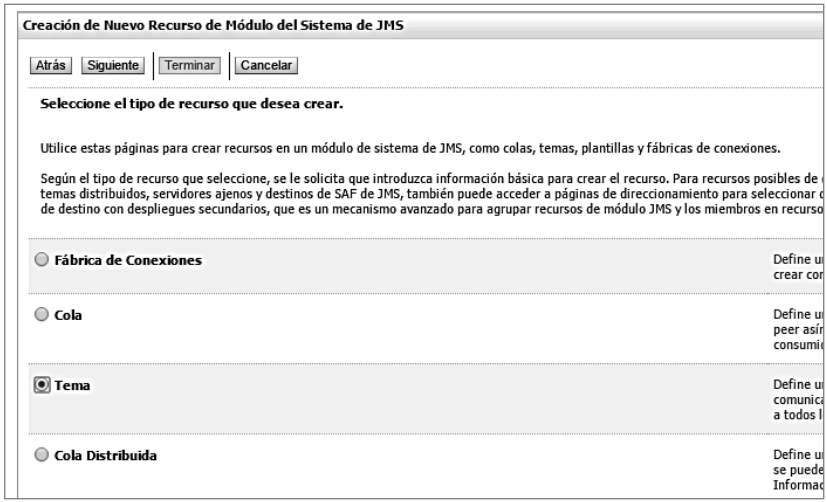
Le asignamos un nombre y una ruta en el árbol JNDI. En nuestro caso, lo situamos junto a las factorías de conexiones ("javax/jms").



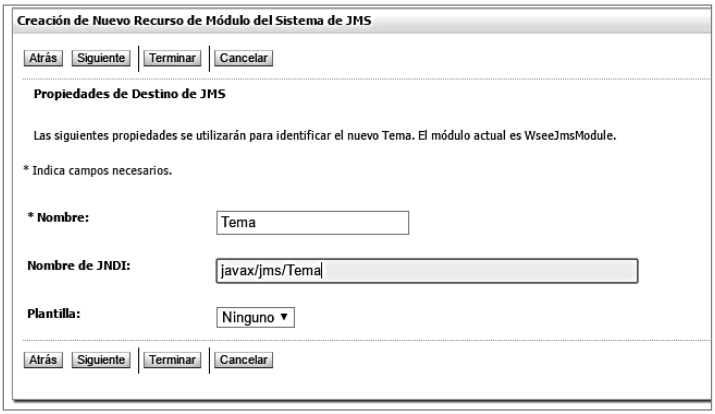
Elegimos el servidor JMS (depende del servidor concreto). En nuestro caso "Wsee-JmsServer".



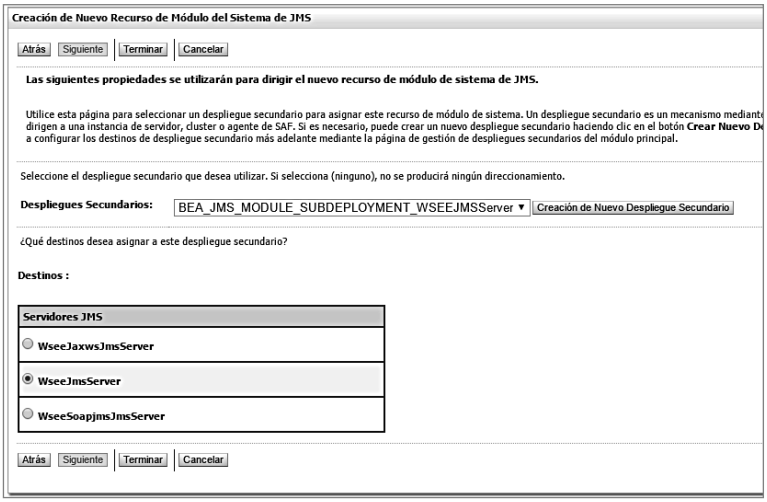
Aprovechando que estamos en la consola de administración, crearemos también el canal para las conexiones de tipo tema o *topic*. Crearemos un tipo de recurso "Tema".



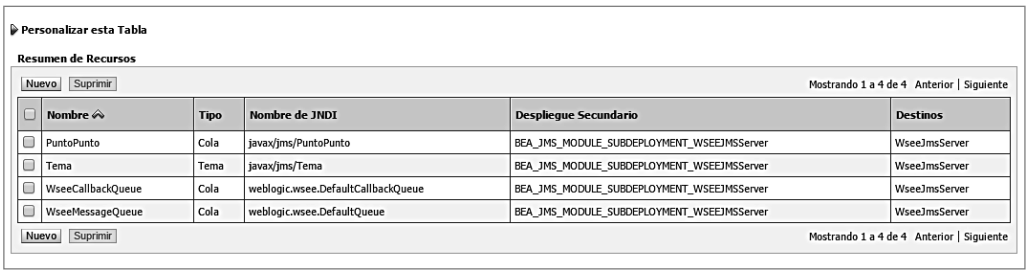
Le asignamos un nombre y una ruta JNDI.



Volvemos a elegir el servidor JMS. En nuestro caso "WseeJmsServer".



Veamos cómo quedan los recursos JMS con los que trabajaremos:



Desarrollaremos una sencilla aplicación empresarial en la que mostraremos una entrada de texto al usuario. El *servlet* que reciba la petición será el productor del mensaje, que lo enviará a través del canal "PuntoPunto" que acabamos de crear y el MDB suscrito al mismo canal será el que lo reciba. Los artefactos *software* de este ejemplo son los siguientes:

- Página "index.jsp" en la que el usuario introducirá su entrada:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <form action="PuntoPuntoServlet" method="post">
      <table>
        <tr>
          <td>Inserte el texto:</td>
          <td><input type="text" name="entrada"
                size="40" maxlength="40" /></td>
          <td><input type="submit" /></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

- "PuntoPuntoServlet.java". Recibe la entrada del usuario, genera el mensaje y lo envía a través del canal, usando la factoría de conexiones:

```
package servlets;
import java.io.IOException;
import java.io.PrintWriter;
import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
public class PuntoPuntoServlet extends HttpServlet {
   @Resource(mappedName = "javax.jms.PuntoPunto")
   private Queue puntoPunto;
   @Resource(mappedName = "javax.jms.QueueConnectionFactory")
   private ConnectionFactory connectionFactory;
   private Connection connection = null;
   private Session session = null;
   private MessageProducer producer = null;
   private Message message = null;
   private boolean ok = false;
   protected void processRequest(HttpServletRequest request,
           HttpServletResponse response)
       throws ServletException, IOException {
      String entrada = request.getParameter("entrada");
      try {
         sendJMSMessageToPuntoPunto(entrada);
         ok = true;
      } catch(Exception e) {
         System.out.println(e.getMessage());
         ok = false;
      }
      response.setContentType("text/html;charset=UTF-8");
      try (PrintWriter out = response.getWriter()) {
         out.println("<!DOCTYPE html>");
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet MiServlet</title>");
         out.println("</head>");
         out.println("<body>");
         if (ok) {
            out.println("<h1>El mensaje " + entrada +
                  " se envió al MDB...</h1>");
         } else {
            out.println("<h1>No se pudo enviar el mensaje " + entrada +
                  " al MDB...</h1>");
         }
         out.println("</body>");
         out.println("</html>");
      }
   }
   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      processRequest(request, response);
   }
```

```
  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    processRequest(request, response);
  }
  @Override
  public String getServletInfo() {
    return "Información del servlet";
  }
  private void sendJMSMessageToPuntoPunto(String mensaje)
      throws JMSException {
    setMessageProducer();
    message = createJMSStringMessage(session, mensaje);
    producer.send(message);
  }
  private Message createJMSStringMessage(Session session, String text)
      throws JMSException {
    TextMessage msg = session.createTextMessage();
    msg.setText(text);
    return msg;
  }
  private void setMessageProducer() throws JMSException {
    if (producer == null) {
      connection = connectionFactory.createConnection();
      session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
      producer = session.createProducer(puntoPunto);
    }
  }
}
```

- Con las líneas:

```
@Resource(mappedName = "javax.jms.PuntoPunto")
private Queue puntoPunto;
@Resource(mappedName = "javax.jms.QueueConnectionFactory")
private ConnectionFactory connectionFactory;
```

Realizamos la llamada (inyección de dependencias) a nuestro canal. El código que se encarga de enviar el mensaje a la cola es el siguiente:

```
private void sendJMSMessageToPuntoPunto(String mensaje)
    throws JMSException {
  setMessageProducer();
  message = createJMSStringMessage(session, mensaje);
  producer.send(message);
}
private Message createJMSStringMessage(Session session, String text)
    throws JMSException {
```

```
    TextMessage msg = session.createTextMessage();
    msg.setText(text);
    return msg;
}
private void setMessageProducer() throws JMSException {
    if (producer == null) {
        connection = connectionFactory.createConnection();
        session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
        producer = session.createProducer(puntoPunto);
    }
}
```

Para enviar el mensaje, primero creamos el productor, preparamos el mensaje y acto seguido lo enviamos. En dicho código llama la atención la instrucción en la que creamos la sesión: "session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);". A la instrucción "createSession" le pasamos dos parámetros. El primero sirve como una instrucción *begin* de una transacción, en la que posteriormente se deberá ejecutar *commit* o *rollback (true)*; o si el propio canal se encarga de administrar los mensajes *(false)*. La constante "Session.CLIENT_ACKNOWLEDGE" indica un método de acuse de recibo en el que el cliente reconoce un mensaje consumido llamando al método "acknowledge()".

Los posibles valores para esta constante son:

- Session.AUTO_ACKNOWLEDGE.
- Session.CLIENT_ACKNOWLEDGE.
- Session.DUPS_OK_ACKNOWLEDGE.
- Session.SESSION_TRANSACTED.

Se deja al lector profundizar sobre este asunto.

- El *bean* "PuntoPuntoBean" queda como sigue:

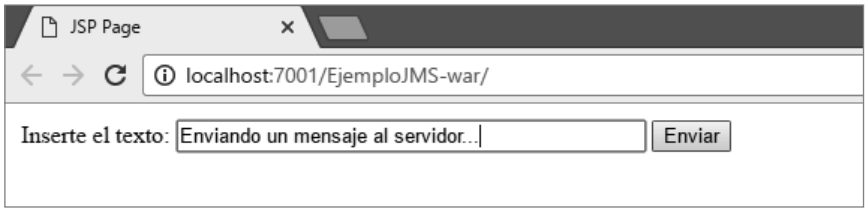
```
package beans;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "javax.jms.PuntoPunto"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "javax.jms.Queue")
})
public class PuntoPuntoBean implements MessageListener {
    public PuntoPuntoBean() {}
```

```
   @Override
   public void onMessage(Message message) {
      TextMessage msg = (TextMessage) message;
      String text = "";
      try {
         text = msg.getText();
      } catch (Exception e) {}
      System.out.println("En el bean tenemos: "+text+"".);
   }
}
```

Vemos cómo se configura el *bean* mediante la anotación:

```
@MessageDriven(activationConfig = {
   @ActivationConfigProperty(propertyName = "destinationLookup",
         propertyValue = "javax.jms.PuntoPunto"),
   @ActivationConfigProperty(propertyName = "destinationType",
         propertyValue = "javax.jms.Queue")
})
```

La recepción se realiza en el método "onMessage", de forma asíncrona, ejecutable cada vez que se establezca un mensaje en la cola.



La salida en la consola del sistema del servidor es la siguiente:

En el bean *tenemos: enviando un mensaje al servidor...*

Dejamos al lector que realice la prueba de poner otro *bean* a la escucha del canal de tipo cola. Como nota, y recordando que dicho canal es para un emisor y un receptor, mostramos la salida por consola:

En el otro bean *tenemos: mensaje en la cola...*

La entrega del mensaje se realiza únicamente a un *bean*.

Para finalizar con los MDB, y siguiendo la misma filosofía, veamos el código de un *bean* de tipo *topic* ("Tema"). Pondremos un *servlet* emisor y dos *beans* suscritos a su tema.

- *Servlet* "TemaServlet.java":

```
package servlets;
import java.io.IOException;
import java.io.PrintWriter;
import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class TemaServlet extends HttpServlet {
   @Resource(mappedName = "javax.jms.Tema")
   private Topic tema;
   @Resource(mappedName = "javax.jms.TopicConnectionFactory")
   private ConnectionFactory connectionFactory;
   private Connection connection = null;
   private Session session = null;
   private MessageProducer producer = null;
   private Message message = null;
   private boolean ok = false;
   protected void processRequest(HttpServletRequest request,
         HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      String entrada = request.getParameter("entrada");
      try {
         sendJMSMessageToTema(entrada);
         ok = true;
      } catch (Exception e) {
         ok = false;
      }
      try (PrintWriter out = response.getWriter()) {
         out.println("<!DOCTYPE html>");
         out.println("<html>");
         out.println("<head>");
```

```
            out.println("<title>Servlet TemaServlet</title>");
            out.println("</head>");
            out.println("<body>");
            if(ok) {
               out.println("<h1>El mensaje " + entrada +
                     " se envió al MDB...</h1>");
            } else {
               out.println("<h1>El mensaje " + entrada +
                     " no se pudo enviar al MDB...</h1>");
            }
            out.println("</body>");
            out.println("</html>");
         }
      }
      @Override
      protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
         processRequest(request, response);
      }
      @Override
      protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
         processRequest(request, response);
      }
      @Override
      public String getServletInfo() {
         return "Información del servlet";
      }
      private void sendJMSMessageToTema(String mensaje) throws JMSException {
         setMessageProducer();
         message = createJMSStringMessage(session, mensaje);
         producer.send(message);
      }
      private Message createJMSStringMessage(Session session,
            String text) throws JMSException {
         TextMessage msg = session.createTextMessage();
         msg.setText(text);
         return msg;
      }
      private void setMessageProducer() throws JMSException {
         if (producer == null) {
            connection = connectionFactory.createConnection();
            session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
            producer = session.createProducer(tema);
         }
      }
}
```

La filosofía es la misma. Veamos los dos *beans* que vamos a tener a la escucha:

- "TemaBean.java":

```
package beans;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
@MessageDriven(activationConfig = {
   @ActivationConfigProperty(propertyName = "clientId",
         propertyValue = "javax/jms/Tema"),
   @ActivationConfigProperty(propertyName = "destinationLookup",
         propertyValue = "javax/jms/Tema"),
   @ActivationConfigProperty(propertyName = "subscriptionDurability",
         propertyValue = "Durable"),
   @ActivationConfigProperty(propertyName = "subscriptionName",
         propertyValue = "javax/jms/Tema"),
   @ActivationConfigProperty(propertyName = "destinationType",
         propertyValue = "javax.jms.Topic")
})
public class TemaBean implements MessageListener {
   public TemaBean() {
   }
   @Override
   public void onMessage(Message message) {
      TextMessage msg = (TextMessage) message;
      String text = "";
      try {
         text = msg.getText();
      } catch (Exception e) {}
      System.out.println("En tema bean tenemos: "+text+"".);
   }
}
```

- *Bean* "OtroTemaBean.java":

```
package beans;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(activationConfig = {
   @ActivationConfigProperty(propertyName = "clientId",
         propertyValue = "javax/jms/Tema2"),
   @ActivationConfigProperty(propertyName = "destinationLookup",
         propertyValue = "javax/jms/Tema"),
   @ActivationConfigProperty(propertyName = "subscriptionDurability",
         propertyValue = "Durable"),
   @ActivationConfigProperty(propertyName = "subscriptionName",
         propertyValue = "javax/jms/Tema"),
```

```
  @ActivationConfigProperty(propertyName = "destinationType",
      propertyValue = "javax.jms.Topic")
})
public class OtroTemaBean implements MessageListener {
  public OtroTemaBean() {
  }
  @Override
  public void onMessage(Message message) {
    TextMessage msg = (TextMessage) message;
    String text = "";
    try {
      text = msg.getText();
    } catch (Exception e) {}
    System.out.println("En otro tema bean tenemos: "+text+"".);
  }
}
```

La anotación del *bean* se realiza con el siguiente código:

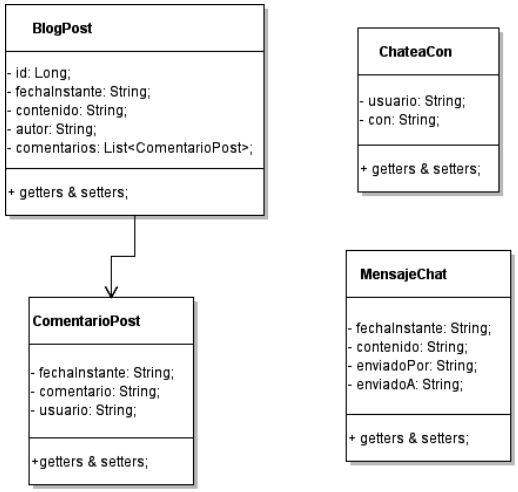
```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(propertyName = "clientId",
      propertyValue = "javax/jms/Tema"),
  @ActivationConfigProperty(propertyName = "destinationLookup",
      propertyValue = "javax/jms/Tema"),
  @ActivationConfigProperty(propertyName = "subscriptionDurability",
      propertyValue = "Durable"),
  @ActivationConfigProperty(propertyName = "subscriptionName",
      propertyValue = "javax/jms/Tema"),
  @ActivationConfigProperty(propertyName = "destinationType",
      propertyValue = "javax.jms.Topic")
})
```

Dejamos al lector investigar sobre estos valores (aunque son bastante intuitivos). La salida por consola es la siguiente:

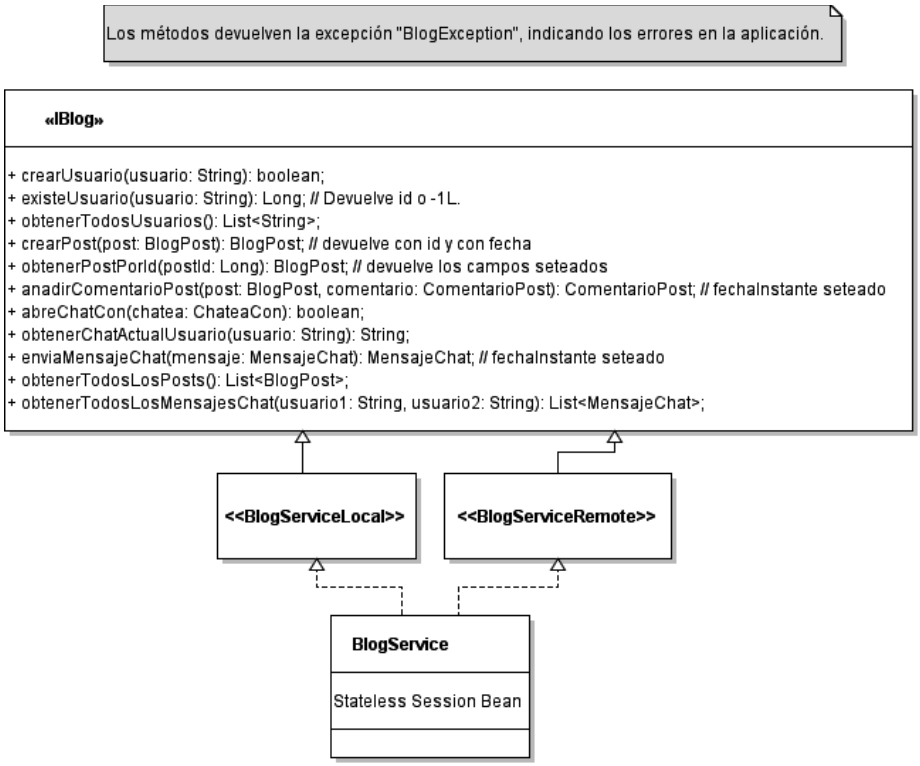
En tema bean *tenemos: enviando mensaje a canal tipo tema...*
En otro tema bean *tenemos: enviando mensaje a canal tipo tema...*

5.5 Ejercicio 4

Como ejercicio, se plantea desarrollar la capa de servicio de la aplicación para crear consistencia en el *back-end* de nuestra aplicación. La capa de servicio trabajará con los siguientes objetos del modelo del dominio (una capa superior al modelo de las entidades).



Y para convertir dichos objetos del dominio en entidades de la base de datos, necesitamos una API en la capa de servicio. Dicha API queda diseñada de la siguiente forma:



Se plantea al lector la implementación de dicha capa de servicio, creando las clases y métodos de utilidad que considere necesarios.

Implementación de MCV: JSF

CAPÍTULO 6

JSF es una implementación propuesta para gestionar el patrón MVC en aplicaciones web. ¿Qué ocurre si usamos *servlets* y páginas JSP para implementar el controlador y las vistas de nuestra aplicación? No ocurre nada. Lo podemos usar perfectamente. Pero, siempre que se desarrolla un *framework* para una tecnología, se pretende dotar al desarrollador de un conjunto de herramientas que hagan más fácil y menos tediosa la escritura del código, haciendo que se centre mayormente en la lógica de negocio de su aplicación.

Si pretendemos desarrollar una aplicación web usando *servlets* y páginas JSP, debemos estar pendientes de tener siempre correctamente seteados los JavaBeans de sesión o de aplicación que vimos en el capítulo correspondiente. Cada acción del usuario que afecte al modelo de nuestra aplicación debe enviarse a un *servlet*; éste será el encargado de procesar la petición, hacer la llamada correspondiente a la capa de servicio, tratar la respuesta del *back-end* (incluidas las excepciones) y setear correctamente los JavaBeans antes de despachar la vista (página JSP) correspondiente. Ya que tenemos componentes *software* en el *back-end* (EJB), ¿por qué no tenemos también componentes de la capa de usuario que sean fáciles de administrar? Ésta es precisamente la idea de JSF (acrónimo de *Java Server Faces*: Vistas del Servidor Java). Dicha tecnología pretende que la interfaz de usuario esté compuesta por un conjunto de componentes. Cada componente tiene lo que llamaríamos un *bean de respaldo* o *bean gestionado (managed bean)*. La versión actual de JSF es la 2.2 e incluye una serie de características:

- Conjunto de componentes de la interfaz de usuario.
- Librerías de etiquetas para hacer más fácil la escritura y lectura del código JSP, evitando los *scriptlets*.
- *Beans* administrados.

- Modelo de eventos.
- Soporte para HTML5/CSS3.

Y muchas más características. El mundo de JSF requiere un manual en sí, por lo tanto, nosotros nos centraremos en las características generales de JSF. Además, el propio JSF tiene varias subimplementaciones, que hacen mucho más rico el conjunto de componentes que podemos usar. Estas subimplementaciones del *framework* son las siguientes:

- RichFaces.
- OpenFaces.
- PrimeFaces.
- jQuery4Jsf.
- IceFaces.

Cada cual le aporta su riqueza al *framework*. Nosotros nos centraremos principalmente en JSF en sí, pero es un maravilloso mundo en el que el lector puede adentrarse para desarrollar interfaces de usuario profesionales de sus aplicaciones empresariales. Centrémonos en estudiar las generalidades de JSF.

¿Por qué es JSF una implementación de MVC? Porque actúa como núcleo de dicho patrón. El modelo estará representado por los *stateless session beans* que residirán en el contenedor de EJB. Con dichos EJB seremos capaces de crear una API sin estado que podremos llamar desde el controlador. En la arquitectura de *servlets* + JSP, el *servlet* es el controlador y las páginas JSP son las vistas. Dicha tupla *servlet* + JSP se ha convertido ahora en la tupla componente + *managed bean*.

6.1 *Managed beans*

Los *beans* administrados son los que nos permiten setear las propiedades que queremos mostrar al usuario, esto es, aquellos datos con los que trabajarán los componentes UI *(User Interface)*. Al igual que ocurría con los JavaBeans, los *managed beans* tienen sus ámbitos de existencia o ciclos de vida (aquellos en los que las variables permanecen inalteradas para el usuario). Veamos los ámbitos de estos *beans*:

- ***Dependent.*** Su ciclo de vida depende del ciclo de vida de otro *bean* (seteado con la anotación "@ManagedProperty").
- ***View.*** El ciclo de vida del *managed bean* permanece mientras se muestre en pantalla la misma página JSF.
- ***Application.*** Su ciclo de vida se corresponde con el de la aplicación. Estos *beans* contendrán datos que serán iguales para todos los usuarios de la aplicación (di-

ferentes sesiones). Su ámbito se corresponde con el de los *Application JavaBeans* que vimos en su sección correspondiente.

- ***Request.*** Su ciclo de vida se equipara al ciclo de vida de la petición o *request*. Esto es, desde que el cliente HTTP lanza la petición por el método GET o el método POST hasta que la respuesta es enviada al cliente. Estos tipos de *managed beans* nos pueden venir bien para setear campos de formularios que son enviados al servidor.
- ***Session.*** Su ciclo de vida se corresponde con el de la sesión. Serán los más utilizados, ya que cada cliente HTTP tendrá su propio conjunto de variables propias de la aplicación mientras dure la sesión del usuario.
- ***Conversation.*** El ciclo de vida del *bean* dura mientras se ejecute el inicio y el fin de una conversación, bien definida con los métodos *begin* y *end*.
- ***Flow.*** Este tipo de *beans* se usa para la navegación entre las páginas.

En nuestros ejemplos estudiaremos de forma aplicada los *beans* cuyo ámbito de acceso es la sesión *(SessionScoped)*.

6.2 Librerías de etiquetas en las páginas JSF

Tenemos tres librerías de etiquetas. A cada una la referenciamos con un prefijo:

- "h:". Es el conjunto de etiquetas de HTML definido sobre JSF.
- "f:". *Core* de JSF. Es un conjunto de etiquetas que ofrecen funcionalidad extra al código JSF.
- "ui:". Librería de Facelets. Nos permite definir plantillas.

Veamos el conjunto de etiquetas de cada librería. No entraremos a estudiarlas una por una, pero sí estudiaremos de forma aplicada aquellas que usemos en nuestro código.

Conjunto de etiquetas en la librería de HTML:

- h:commandButton.
- h:commandLink.
- h:dataTable.
- h:form.
- h:graphicImage.
- h:inputFile.
- h:inputHidden.
- h:inputSecret.
- h:inputText.

- h:inputTextarea.
- h:message.
- h:messages.
- h:button.
- h:link.
- h:body.
- h:doctype.
- h:outputFormat.
- h:head.
- h:outputLabel.
- h:outputLink.
- h:outputText.
- h:outputScript.
- h:outputStylesheet.
- h:panelGrid.
- h:panelGroup.
- h:panelpassthrough.Element.
- h:selectBooleanCheckbox.
- h:selectManyCheckbox.
- h:selectManyListbox.
- h:selectManyMenu.
- h:selectOneListbox.
- h:selectOneMenu.
- h:selectOneRadio.
- h:column.

Conjunto de etiquetas en la librería *core* (aportan funcionalidad extra):

- f:actionListener.
- f:ajax.
- f:attribute.
- f:attributes.
- f:passThroughAttribute.
- f:passThroughAttributes.

- f:converter.
- f:convertDateTime.
- f:convertNumber.
- f:event.
- f:facet.
- f:loadBundle.
- f:metadata.
- f:param.
- f:phaseListener.
- f:selectItem.
- f:selectItems.
- f:setPropertyActionListener.
- f:subview.
- f:validateDoubleRange.
- f:validateLength.
- f:validateLongRange.
- f:validateBean.
- f:validateRegex.
- f:validateRequired.
- f:validator.
- f:valueChangeListener.
- f:verbatim.
- f:view.
- f:viewParam.
- f:viewAction.
- f:resetValues.

Conjunto de etiquetas en la librería "ui" (Facelets, definición de plantillas):

- ui:component.
- ui:composition.
- ui:debug.
- ui:define.
- ui:decorate.

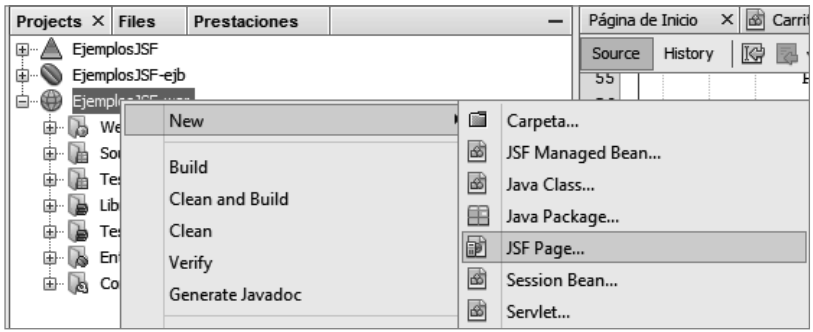
- ui:fragment.
- ui:include.
- ui:insert.
- ui:param.
- ui:repeat.
- ui:remove.

Referencia para ver las etiquetas: "http://docs.oracle.com/javaee/7/".

6.3 Renderizando componentes

Como hemos indicado, cada componente debe tener un *bean* de respaldo. Hay algunos componentes que servirán para maquetar y diseñar nuestras vistas. Dichos componentes deben tener, evidentemente, un *bean* de respaldo. El componente concreto usará un booleano en el *bean* para saber si se renderiza (se muestra) o no.

Una página JSF no es más que una página JSP en la que importamos las librerías que anteriormente hemos citado, haciendo uso de ellas con los prefijos indicados. Veamos un primer ejemplo de página JSF, con una imagen en la que mostramos cómo crear dicha página en NetBeans. A dicha página la llamamos *index.xhtml* (ésta es la extensión de las páginas JSF).

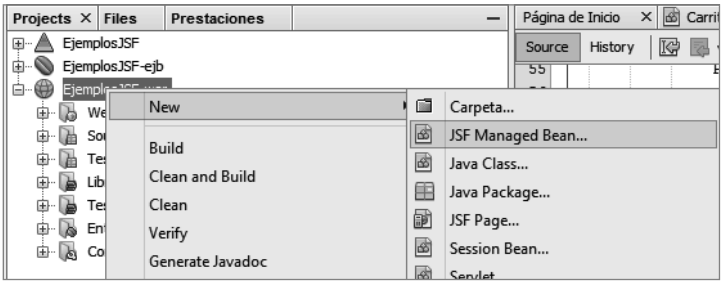


El código de nuestra página JSF es el siguiente, en el que podemos ver cómo importamos las librerías del apartado anterior:

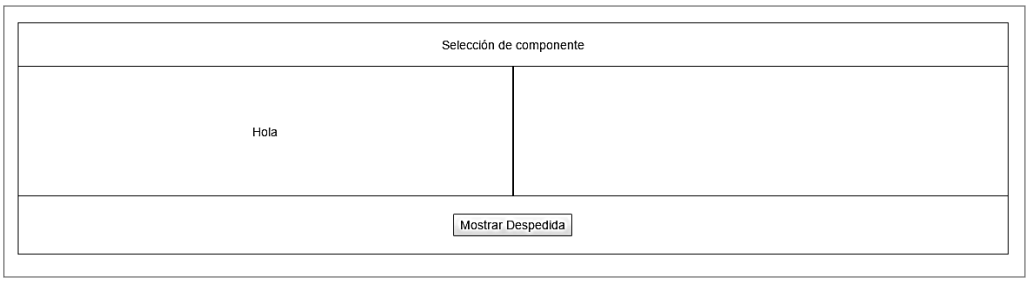
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
```

```
  <h:head>
    <title>Página JSF</title>
  </h:head>
  <h:body>
    <h1>Hola desde JSF</h1>
  </h:body>
</html>
```

Empecemos a unir las páginas JSF con los *beans* administrados. Primero, veamos cómo podemos introducir un *managed bean*:



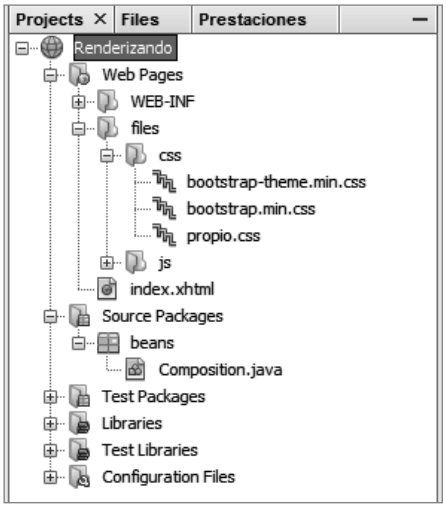
Ahora que ya sabemos cómo incluir en nuestro proyecto páginas JSF y *beans* administrados, pensemos en una pequeña aplicación en la que tendremos un botón que setee si se muestra un saludo o una despedida. El texto que muestra dicho saludo o despedida será un componente en sí. De esta forma tan simple sabremos cómo renderizar componentes. La imagen de la vista podría ser ésta:



Cada vez que pulsemos en el botón "Mostrar Saludo" o "Mostrar Despedida", se modificará el estado del *bean* gestionado. En dicho *bean* tendremos dos booleanos (aunque para este caso con uno nos bastaría). En el *bean* también guardamos el texto que mostramos en el botón. Para este pequeño proyecto tendremos:

- Una hoja de estilos CSS para dar dicha composición a la pantalla.
- El *bean* administrado que incluirá el código de la acción de cambio de estado.
- La página JSF "index.xhtml" que incluye el componente.

Veamos cuál sería la estructura del proyecto.



Los tres elementos los podemos ver en los ficheros "propio.css", "index.xhtml" y "Composition.java". Veamos cada uno de ellos:

- Hoja de estilos "propio.css":

```
html, body {
   height: 100%;
}
div.container {
   margin-top: 1cm;
}
div.container div.header {
   height: 50px;
   line-height: 50px;
   text-align: center;
   border-left: 1px solid black;
   border-top: 1px solid black;
   border-right: 1px solid black;
}
div.container div.panelcomponent {
   height: 150px;
   line-height: 150px;
   border: 1px solid black;
   text-align: center;
}
div.container div.formbutton {
   text-align: center;
   border-left: 1px solid black;
```

```
   border-bottom: 1px solid black;
   border-right: 1px solid black;
   margin-bottom: 1cm;
}
div.container div.formbutton input {
   margin: 20px;
}
```

El código del *bean* administrado "Composition.java" es el siguiente:

```
package beans;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
import javax.faces.bean.ManagedBean;
@ManagedBean
@Named(value = "composition")
@SessionScoped
public class Composition implements Serializable {
   private boolean mostrarSaludo;
   private boolean mostrarDespedida;
   private String texto = "";
   public Composition() {
      this.mostrarSaludo = true;
      this.mostrarDespedida = false;
      this.texto = "Mostrar Despedida";
   }
   public void cambiaEstado() {
      if(this.mostrarSaludo) {
         this.mostrarSaludo = false;
         this.mostrarDespedida = true;
         this.texto = "Mostrar Saludo";
      } else {
         this.mostrarSaludo = true;
         this.mostrarDespedida = false;
         this.texto = "Mostrar Despedida";
      }
   }
   public boolean isMostrarSaludo() {
      return mostrarSaludo;
   }
   public void setMostrarSaludo(boolean mostrarSaludo) {
      this.mostrarSaludo = mostrarSaludo;
   }
   public boolean isMostrarDespedida() {
      return mostrarDespedida;
   }
```

```
  public void setMostrarDespedida(boolean mostrarDespedida) {
    this.mostrarDespedida = mostrarDespedida;
  }
  public String getTexto() {
    return texto;
  }
  public void setTexto(String texto) {
    this.texto = texto;
  }
}
```

Debemos pararnos a observar cómo hemos usado las anotaciones "@ManagedBean", "@SessionScoped" y "@Named(value = "composition")". Con dichas anotaciones indicamos que es un *bean* administrado (de respaldo), que su ámbito es la sesión y que el nombre con el que lo vamos a referenciar desde las páginas JSF es "composition". El funcionamiento de la aplicación queda definido en el método "cambiaEstado()" que se lanza desde la página JSF. El código de "index.xhtml" es el siguiente:

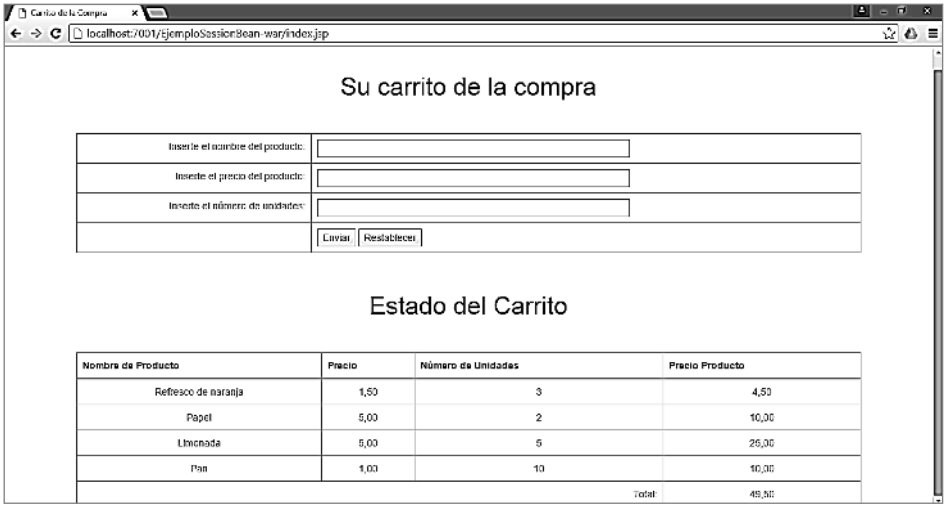
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" href="files/css/bootstrap.min.css" />
    <link rel="stylesheet" type="text/css" href="files/css/bootstrap-theme.min.css" />
    <link rel="stylesheet" type="text/css" href="files/css/propio.css" />
    <script type="text/javascript" src="files/js/jquery-3.1.0.min.js"></script>
    <script type="text/javascript" src="files/js/bootstrap.min.js"></script>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Renderizando en JSF</title>
  </h:head>
  <h:body>
    <div class="container">
      <div class="header col-md-12">
        <h:outputText value="Selección de componente" />
      </div>
      <div class="panelcomponent col-md-6">
        <h:outputText value="Hola" rendered="#{composition.mostrarSaludo}" />
      </div>
      <div class="panelcomponent col-md-6 second">
        <h:outputText value="Adiós" rendered="#{composition.mostrarDespedida}" />
      </div>
```

```
        <div class="formbutton col-md-12">
          <h:form>
            <h:commandButton value="#{composition.texto}" type="submit"
                    action="#{composition.cambiaEstado()}" />
          </h:form>
        </div>
      </div>
    </h:body>
</html>
```

El lector se habrá percatado de que este *bean* de respaldo no se corresponde con un componente definido mediante una etiqueta de JSF, sino que se corresponde con la composición completa. Podríamos decir, no de forma del todo correcta, que se corresponde con el "div" de la clase "container".

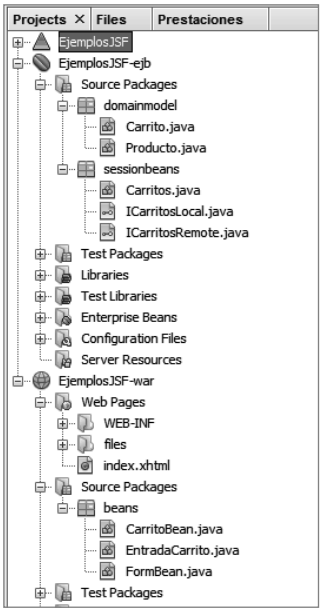
6.4 Ejemplo del carrito de la compra

Un buen ejemplo para empezar a practicar JSF puede ser el del carrito de la compra que vimos en el capítulo de EJB. Recordemos la presentación (la vista) de la única pantalla de esta aplicación:



Lo primero que tenemos que preguntarnos es: ¿cuántos componentes vamos a tener? Está claro que la respuesta a esta pregunta nos responde a otra: ¿cuántos *beans* de respaldo vamos a tener? Pero, antes de todo, debemos preguntarnos: ¿la composición general requiere un componente? La composición general requeriría un componente si ésta fuera dinámica, es decir: la composición está compuesta por un formulario y una tabla. ¿En algún momento dicha composición puede variar? La respuesta es no.

Por lo tanto, nuestra composición general no requiere de un componente. Bajando de nivel nos encontramos dos componentes: un formulario y una tabla. De modo que ya sabemos la respuesta a cuántos componentes vamos a tener: dos. Y, por lo tanto, ya sabemos cuántos *managed beans* (*beans* de respaldo) vamos a tener: dos. Veamos la estructura del proyecto:



Los dos *beans* de respaldo tendrán:

- **El del formulario.** Un listado de campos con las entradas del carrito.
- **El del estado del carrito.** Un listado de productos.

Por lo tanto, en cada *managed bean* habrá un listado. Necesitamos definir una clase que tenga los tres campos con la entrada del usuario. Dicha clase podría ser "EntradaCarrito.java".

- "EntradaCarrito.java":

```
package beans;
import java.io.Serializable;
public class EntradaCarrito implements Serializable {
   private String text;
   private String value;
   public EntradaCarrito(){
      this.text = "";
      this.value = "";
   }
   public String getText() {
      return text;
```

```
    }
    public void setText(String text) {
        this.text = text;
    }
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }
}
```

Con el conocimiento que ya tenemos, podemos afirmar que es un JavaBean. ¿Y el listado de productos? Podemos usar perfectamente el *bean* de "Producto" definido en el módulo de EJB.

- "Producto.java":

```
package domainmodel;
import java.io.Serializable;
public class Producto implements Serializable{
    private String nombre;
    private double precio;
    private int nUnidades;
    private double precioTotal;
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public double getPrecio() {
        return precio;
    }
    public void setPrecio(double precio) {
        this.precio = precio;
    }
    public int getnUnidades() {
        return nUnidades;
    }
    public void setnUnidades(int nUnidades) {
        this.nUnidades = nUnidades;
    }
    public double getPrecioTotal() {
        return precioTotal;
    }
    public void setPrecioTotal(double precioTotal) {
        this.precioTotal = precioTotal;
    }
}
```

Debemos indicar que hemos añadido, con respecto al ejemplo que presentamos en el capítulo de EJB, el campo "precioTotal" que tiene el producto (multiplicación) por el número de unidades de dicho producto. ¿Por qué lo hacemos en el *bean*? Para evitar hacer operaciones en la página JSF. Por ello mismo, modificamos la lógica de negocio cada vez que añadimos un producto al carrito, seteando dicho campo (también en el módulo de EJB).

- "Carrito.java":

```
package domainmodel;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
public class Carrito implements Serializable {
   private List<Producto> productos;
   private double valorTotal;
   public Carrito() {
      this.productos = new ArrayList<Producto>();
   }
   public List<Producto> getProductos() {
      return productos;
   }
   public void setProductos(List<Producto> productos) {
      this.productos = productos;
   }
   public double getValorTotal() {
      return valorTotal;
   }
   public void setValorTotal(double valorTotal) {
      this.valorTotal = valorTotal;
   }
   public void anadirProductoCarrito(Producto p) {
      if ((p != null) && (this.productos != null)) {
         boolean encontrado = false;
         int i = 0;
         for (Producto producto : this.productos) {
            if (producto.getNombre().equals(p.getNombre())) {
               encontrado = true;
               break;
            } else {
               i++;
            }
         }
         if (encontrado) {
            Producto esteProducto = this.productos.get(i);
            esteProducto.setnUnidades(esteProducto.getnUnidades()
                  + p.getnUnidades());
```

```
            double estePrecioTotal =
               (double)esteProducto.getnUnidades() * esteProducto.getPrecio();
            esteProducto.setPrecioTotal(estePrecioTotal);
         } else {
            double estePrecioTotal = (double)p.getnUnidades() * p.getPrecio();
            p.setPrecioTotal(estePrecioTotal);
            this.productos.add(p);
         }
      }
   }
   public void eliminarProductoCarrito(Producto p) {
      if ((p != null) && (this.productos != null)) {
         boolean encontrado = false;
         int i = 0;
         for (Producto producto : this.productos) {
            if (producto.getNombre().equals(p.getNombre())) {
               encontrado = true;
               break;
            } else {
               i++;
            }
         }
         if (encontrado) {
            this.productos.remove(i);
         }
      }
   }
   public double getPrecioTotalCarrito() {
      if (this.productos != null) {
         double total = 0.0;
         for (Producto p : this.productos) {
            total = total + (p.getPrecio() * p.getnUnidades());
         }
         return total;
      } else {
         return 0.0;
      }
   }
}
```

Excepto este ajuste en el módulo de EJB, todo el código de dicho módulo permanece inalterado. Las interfaces locales y remotas permanecen igual y la implementación del EJB *singleton* también permanece tal y como vimos en el capítulo de EJB. Pasemos a estudiar nuestros *managed beans* en el módulo WAR (donde también definimos el JavaBean "EntradaCarrito.java" ya visto). El *bean* que va a administrar las entradas del usuario se llamará *FormBean* y será un *bean* de sesión.

- "FormBean.java":

```
package beans;
import javax.inject.Named;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
@ManagedBean
@SessionScoped
@Named(value = "formBean")
public class FormBean implements Serializable {
   private List<EntradaCarrito> entradas;
   public FormBean() {
      entradas = new ArrayList<EntradaCarrito>();
      EntradaCarrito entrada = new EntradaCarrito();
      entrada.setText("Inserte el nombre del producto: ");
      entradas.add(entrada);
      entrada = new EntradaCarrito();
      entrada.setText("Inserte el precio del producto: ");
      entradas.add(entrada);
      entrada = new EntradaCarrito();
      entrada.setText("Inserte el número de unidades: ");
      entradas.add(entrada);
   }
   public List<EntradaCarrito> getEntradas() {
      return entradas;
   }
   public void setEntradas(List<EntradaCarrito> entradas) {
      this.entradas = entradas;
   }
   public void limpiaEntradas() {
      for (EntradaCarrito entrada : entradas) {
         entrada.setValue("");
      }
   }
}
```

Hemos decidido setear en el *bean* también los textos que le mostraremos al usuario. El dato que manejará esta vista será la lista de entradas del carrito (nombre del producto, precio del producto y número de unidades).

Veamos cómo resulta el *bean* administrado del carrito, "CarritoBean.java", también *bean* de sesión:

```
package beans;
```

```
import domainmodel.Producto;
import javax.inject.Named;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;
import javax.servlet.http.HttpSession;
import sessionbeans.ICarritosLocal;
@ManagedBean
@SessionScoped
@Named(value = "carritoBean")
public class CarritoBean implements Serializable {
   @EJB
   private ICarritosLocal carritos;
   private List<Producto> productos;
   private double precioTotal;
   @ManagedProperty(value="#{formBean}")
   private FormBean formBean;
   public CarritoBean() {
      this.productos = new ArrayList<Producto>();
      this.precioTotal = 0.0;
   }
   public void anadeCarrito() {
      String sessionId = this.getSessionId();
      carritos.usarCarrito(sessionId);
      List<EntradaCarrito> entradas = formBean.getEntradas();
      String nombre = entradas.get(0).getValue();
      String precioP = entradas.get(1).getValue();
      String nUnidadesP = entradas.get(2).getValue();
      double precio = 0.0;
      int nUnidades = 0;
      String nombreSinEspacios = "";
      try {
         precio = Double.parseDouble(precioP);
         nUnidades = Integer.parseInt(nUnidadesP);
         nombreSinEspacios = nombre.trim();
      } catch (Exception e) {}
      if (!nombreSinEspacios.equals("") && precio != 0.0 && nUnidades != 0) {
         Producto p = new Producto();
         p.setNombre(nombre);
         p.setPrecio(precio);
         p.setnUnidades(nUnidades);
```

```
            carritos.anadirProductoCarrito(sessionId, p);
        }
        formBean.limpiaEntradas();

        this.productos = carritos.getProductos(sessionId);
        this.precioTotal = carritos.getPrecioTotalCarrito(sessionId);
    }
    public void setFormBean(FormBean formBean) {
        this.formBean = formBean;
    }
    public List<Producto> getProductos() {
        return productos;
    }
    public void setProductos(List<Producto> productos) {
        this.productos = productos;
    }
    public double getPrecioTotal() {
        return precioTotal;
    }
    public void setPrecioTotal(double precioTotal) {
        this.precioTotal = precioTotal;
    }
    private String getSessionId() {
        FacesContext fCtx = FacesContext.getCurrentInstance();
        HttpSession session = (HttpSession) fCtx.getExternalContext().getSession(false);
        String sessionId = session.getId();
        return sessionId;
    }
}
```

En este código debemos apreciar:

- **Cómo inyectamos el *bean singleton*.** Podemos hacer la inyección de EJB como si fuera un *servlet*. Gracias a esta anotación, tendremos en el *bean* de respaldo acceso al *back-end* de nuestra aplicación.
- **Cómo inyectamos el *bean* del formulario.** Necesitamos tener acceso al *bean* del formulario para poder procesar los valores introducidos por el usuario. Indicar que se necesita un *setter* del *bean* administrado.
- **Cómo conseguimos el "id" de la sesión.** Haciendo uso de la fachada "FacesContext" conseguimos la instancia actual de la página JSF y con ella podemos tener acceso al "id" de la sesión en la forma indicada en el método "getSessionId". Dicho "id" es necesario para enviárselo al EJB.

Para finalizar, veamos cómo encaja toda nuestra aplicación en la página JSF "index.xhtml". Como nota, indicamos que hacemos uso de los mismos ficheros CSS y JavaScript que usamos en el apartado del carrito visto en el capítulo de EJB, incluyendo el CSS "propio.css".

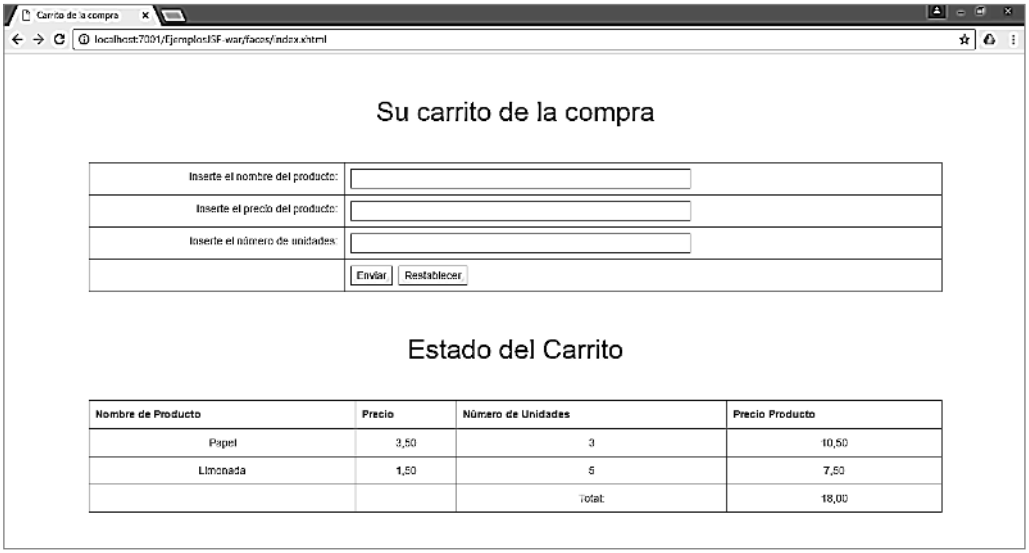
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" href="files/css/bootstrap.min.css" />
    <link rel="stylesheet" type="text/css" href="files/css/bootstrap-theme.min.css" />
    <link rel="stylesheet" type="text/css" href="files/css/propio.css" />
    <script type="text/javascript" src="files/js/jquery-3.1.0.min.js"></script>
    <script type="text/javascript" src="files/js/bootstrap.min.js"></script>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Carrito de la compra</title>
  </h:head>
  <h:body>
    <div class="container">
      <h1>Su carrito de la compra</h1>
    </div>
    <div class="container">
      <h:form>
        <h:dataTable value="#{formBean.entradas}" var="e"
                styleClass="table table-bordered form"
                columnClasses=",input-field"
                footerClass="input-field">
          <h:column>
            <h:outputText value="#{e.text}" />
          </h:column>
          <h:column>
            <h:inputText value="#{e.value}" size="60" maxlength="100" />
            <f:facet name="footer">
              <h:commandButton value="Enviar" type="submit"
                        action="#{carritoBean.anadeCarrito}" />
              <h:outputText value="  " />
              <h:commandButton value="Restablecer" type="reset" />
            </f:facet>
          </h:column>
        </h:dataTable>
      </h:form>
    </div>
```

```
    <div class="container">
      <h1>Estado del Carrito</h1>
    </div>
    <div class="container">
      <h:dataTable value="#{carritoBean.productos}" var="p"
                   styleClass="table table-bordered">
        <h:column>
          <f:facet name="header">
            <h:outputText value="Nombre de Producto" />
          </f:facet>
          <h:outputText value="#{p.nombre}" />
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Precio" />
          </f:facet>
          <h:outputText value="#{p.precio}" >
            <f:convertNumber minFractionDigits="2" />
          </h:outputText>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Número de Unidades" />
          </f:facet>
          <h:outputText value="#{p.getnUnidades()}" />
          <f:facet name="footer">
            <h:outputText value="Total:" />
          </f:facet>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Precio Producto" />
          </f:facet>
          <h:outputText value="#{p.precioTotal}" >
            <f:convertNumber minFractionDigits="2" />
          </h:outputText>
          <f:facet name="footer">
            <h:outputText value="#{carritoBean.precioTotal}" >
              <f:convertNumber minFractionDigits="2" />
            </h:outputText>
          </f:facet>
        </h:column>
      </h:dataTable>
    </div>
  </h:body>
</html>
```

Digno de mención:

- Maquetamos con los contenedores definidos en nuestro CSS.
- Los *managed beans* están disponibles sin necesidad de importarlos.
- Con las librerías de etiquetas <h:> y <f:> evitamos los *scriptlets*. Podemos ver que hemos evitado el "for" del *scriptlet* para todos los elementos del carrito con el componente de tipo "h:dataTable".
- No escribimos en la página directamente, sino a través de la etiqueta de HTML "outputText". La conversión de formato de *double* la realizamos con la etiqueta del *core* de JSF "convertNumber".
- Escribimos la cabecera y el pie de página con la etiqueta <f:facet>.

Veamos finalmente la ejecución del proyecto del carrito en JSF:



6.5 Plantillas con Facelets

La librería que nos permite crear diseños reutilizables que podremos elegir cada vez que los necesitemos es Facelets. Podemos entender Facelets estudiando cuatro etiquetas proporcionadas por su librería:

- <ui:insert>
- <ui:define>
- <ui:include>
- <ui:composition>

Con <ui:insert> definimos bloques de la plantilla que podrán ser reemplazables en posteriores definiciones (sobrescritos por el programador). Imaginemos el siguiente ejemplo de fragmento de plantilla:

```
<h:body>
  <div id="cabecera">
    <ui:insert name="cabecera" >
      Ésta es la cabecera por defecto
    </ui:insert>
  </div>
</h:body>
```

Si tuviésemos una página que heredase de la plantilla anterior (ya veremos cómo), podríamos redefinir el texto de la cabecera de la siguiente forma (usando nuestra segunda etiqueta <ui:define>):

```
<ui:define name="cabecera">
        Ésta es la cabecera sobrescrita
</ui:define>
```

Es decir, la etiqueta <ui:define> sirve para sobrescribir las partes de la plantilla que queramos modificar. En nuestro ejemplo previo, el código que se enviaría al navegador es el siguiente:

```
<h:body>
  <div id="cabecera">
    Ésta es la cabecera sobrescrita
  </div>
</h:body>
```

Con <ui:include> podemos incluir el código que tengamos en otra página JSF. Si tenemos el siguiente código JSF:

```
<h:body>
  <div id="cabecera">
    <ui:insert name="cabecera" >
      <ui:include src="/cabecera.xhtml" />
    </ui:insert>
  </div>
</h:body>
```

Estamos creando un bloque "cabecera":

- Que puede ser sobrescrito con la etiqueta <ui:define>.
- Cuya definición, a menos que sea sobrescrita, se encuentra en el fichero "cabecera.xhtml".

El bloque definido en un fichero externo ("cabecera.xhtml") enlaza con la etiqueta que nos falta por ver: <ui:composition>. El contenido de dicho fichero puede tener esta parte de definición:

```
<body>
  <ui:composition>
    Ésta es la cabecera
  </ui:composition>
</body>
```

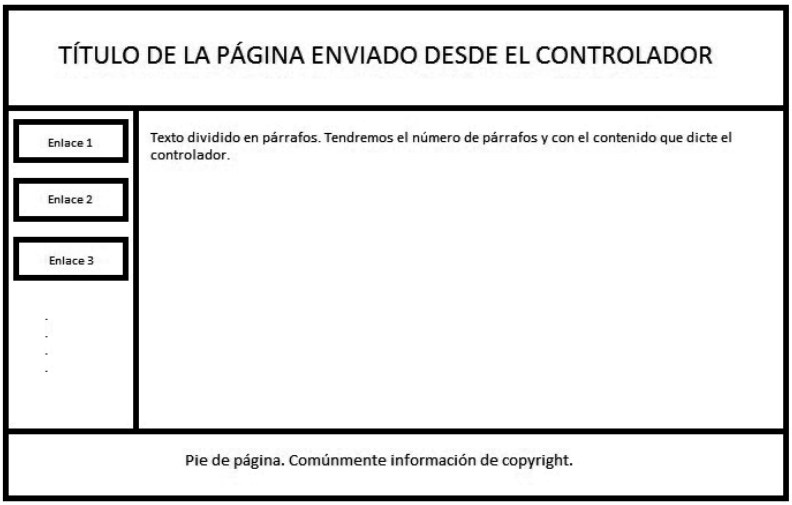
Todo lo que esté fuera del bloque <ui:composition> el motor de JSF no lo tiene en cuenta. De tal forma que nuestro bloque de caso de estudio queda definido (se envía al navegador) de la siguiente forma:

```
<h:body>
  <div id="cabecera">
    Ésta es la cabecera
  </div>
</h:body>
```

Siendo, además, el bloque "cabecera" reemplazable. La etiqueta <ui:composition> es la que nos permite realizar las definiciones externas y la herencia de plantillas. Acabamos de ver un ejemplo de cómo se realizaría una definición externa. ¿Cómo heredaríamos la plantilla? Añadiendo la propiedad *template* a <ui:composition>. Veamos un ejemplo de página que usa la plantilla definida en "plantilla.xhtml".

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <ui:composition template="/plantilla.xhtml">
  </ui:composition>
</html>
```

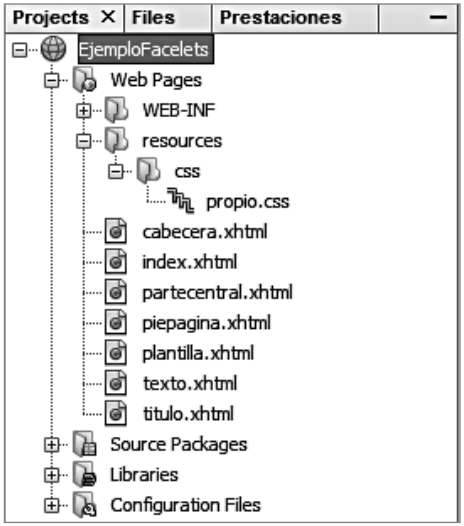
Con este pequeño estudio de las cuatro etiquetas, tenemos el conocimiento suficiente para crear una plantilla en JSF. Imaginemos la siguiente estructura:



Podríamos definir una plantilla con tres bloques:

- Bloque cabecera.
- Bloque parte central.
- Bloque pie de página.

A su vez, los bloques pueden estar definidos en ficheros independientes. Veamos cómo podríamos plantear esta plantilla con Facelets. Una posible estructura del proyecto es la siguiente:



La estructura en árbol de los artefactos JSF es la siguiente, definida mediante <ui:include>:

- plantilla.xhtml.
  - titulo.xhtml.
  - cabecera.xhtml.
  - partecentral.xhtml.
    - texto.xhtml.
  - piepagina.xhtml.

Y la página "index.xhtml" hereda de la plantilla "plantilla.xhtml".

Desde la plantilla importamos un fichero CSS, "propio.css". En esta ocasión hacemos uso de la etiqueta <h:outputStylesheet/>. Veamos los artefactos de código fuente:

- Fichero CSS "propio.css":

```
@charset "UTF-8";
body {
  background-color: gray;
}
div {
  border: 1px solid black;
}
div#contenedor {
  width: 80%;
  margin: 0 auto;
}
div#contenedor div#cabecera {
  text-align: center;
  font-size: xx-large;
  padding: 20px 0;
  color: white;
  background-color: red;
}
div#contenedor div#partecentral div#menu {
  width: 20%;
  float: left;
  border: 0px;
  border-right: 2px solid black;
  text-align: center;
}
div#contenedor div#partecentral div#menu a.boton,
div#contenedor div#partecentral div#menu a.boton:visited {
  margin: 20px;
  display: inline-block;
```

```
   padding: 10px 15px;
   border: 1px solid black;
   text-decoration: none;
   color: white;
   background-color: silver;
}
div#contenedor div#partecentral div#menu a.boton:hover,
div#contenedor div#partecentral div#menu a.boton:active {
   background-color: gray;
}
div#contenedor div#partecentral div#contenido {
   width: 79.3%;
   float: left;
   border: 0px;
   height: auto;
}
div#contenedor div#partecentral div#contenido p {
   margin: 10px;
   text-align: justify;
   color: black;
   font-family: Garamond, Baskerville, "Baskerville Old Face",
      "Hoefler Text", "Times New Roman", serif;
   text-indent: 30px;
}

div#contenedor div#piepagina {
   text-align: center;
   font-size: smaller;
   padding: 10px 0;
   color: white;
   background-color: red;
}
div.clear {
   clear: both;
   border: 0px;
}
```

- Código de la plantilla "plantilla.xhtml":

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
```

```
  <h:head>
    <h:outputStylesheet name="propio.css" library="css" />
    <ui:insert name="title">
      <ui:include src="/titulo.xhtml" />
    </ui:insert>
  </h:head>
  <h:body>
    <div id="contenedor">
      <div id="cabecera">
        <ui:insert name="cabecera" >
          <ui:include src="/cabecera.xhtml" />
        </ui:insert>
      </div>
      <div id="partecentral">
        <ui:insert name="partecentral" >
          <ui:include src="/partecentral.xhtml" />
        </ui:insert>
      </div>
      <div id="piepagina">
        <ui:insert name="piepagina" >
          <ui:include src="/piepagina.xhtml" />
        </ui:insert>
      </div>
    </div>
  </h:body>
</html>
```

Vemos cómo hacemos uso de la hoja de estilos y cómo definimos los bloques que pueden ser sobrescritos en ficheros independientes. Veamos estos ficheros:

- Fichero "titulo.xhtml":

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <body>
    <ui:composition>
      <title>Plantillas</title>
    </ui:composition>
  </body>
</html>
```

- Fichero "cabecera.xhtml":

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://xmlns.jcp.org/jsf/html"
     xmlns:f="http://java.sun.com/jsf/core"
     xmlns:ui="http://java.sun.com/jsf/facelets">
   <body>
      <ui:composition>
         Ésta es la cabecera
      </ui:composition>
   </body>
</html>
```

- Fichero "partecentral.xhtml":

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://xmlns.jcp.org/jsf/html"
     xmlns:f="http://java.sun.com/jsf/core"
     xmlns:ui="http://java.sun.com/jsf/facelets">
   <body>
      <ui:composition>
         <div id="menu">
            <a class="boton" href="#">Enlace 1</a>
            <a class="boton" href="#">Enlace 2</a>
            <a class="boton" href="#">Enlace 3</a>
         </div>
         <div id="contenido">
            <ui:insert name="texto">
               <ui:include src="/texto.xhtml" />
            </ui:insert>
         </div>
         <div class="clear"></div>
      </ui:composition>
   </body>
</html>
```

Apreciamos que se hace inclusión del fichero "texto.xhtml". Veamos el contenido de dicho fichero:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:body>
    <ui:composition>
      <p>Lorem ipsum dolor sit amet.</p>
    </ui:composition>
  </h:body>
</html>
```

Y, finalmente, vemos el fichero del pie de página, "piepagina.xhtml":

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <body>
    <ui:composition>
      Éste es el pie de página
    </ui:composition>
  </body>
</html>
```

Hemos de dejar claro que únicamente hemos definido la plantilla, pero ninguna página. La página "index.xhtml" que hará uso de la plantilla es:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <ui:composition template="/plantilla.xhtml">
    <ui:define name="texto">
      <p>No escribimos Lorem ipsum. Escribimos texto en castellano
        sobrescribiendo la plantilla.</p>
    </ui:define>
  </ui:composition>
</html>
```

A la vez que estamos heredando la plantilla, estamos redefiniendo la sección de texto. El código HTML que se enviaría al navegador es el siguiente:

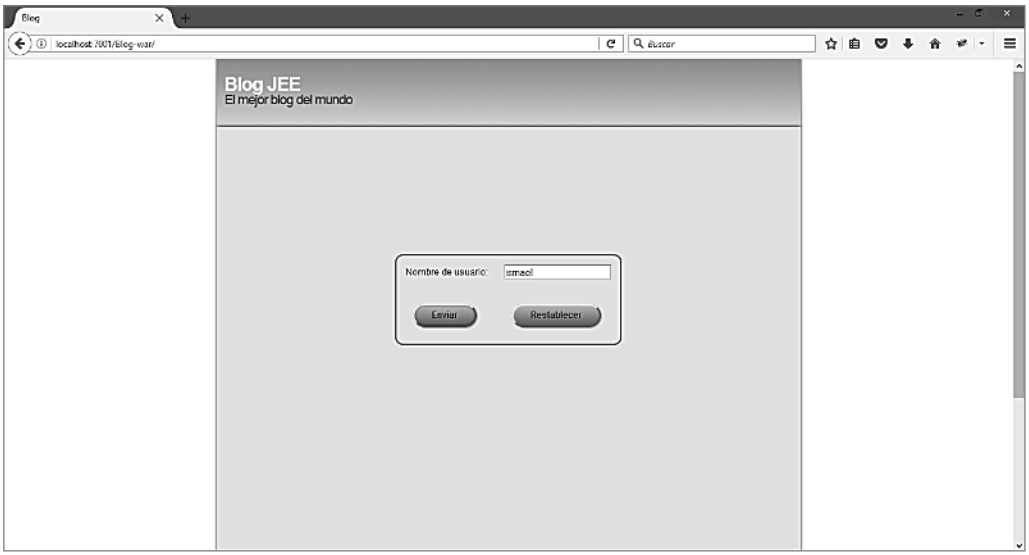
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head id="j_idt2">
    <title>Plantillas</title>
    <link type="text/css" rel="stylesheet"
        href="/EjemploFacelets/faces/javax.faces.resource/propio.css;
        jsessionid=CehYbNqEKfVMq-OP936XJdB-SyvjwlTWwbj5Y-qB9Xqen0HCH-h8
        !1304982310?ln=css" />
  </head>
  <body>
    <div id="contenedor">
      <div id="cabecera">
        Ésta es la cabecera
      </div>
      <div id="partecentral">
        <div id="menu">
          <a class="boton" href="#">Enlace 1</a>
          <a class="boton" href="#">Enlace 2</a>
          <a class="boton" href="#">Enlace 3</a>
        </div>
        <div id="contenido">
          <p>No escribimos Lorem ipsum. Escribimos texto en castellano
            sobrescribiendo la plantilla.</p>
        </div>
        <div class="clear"></div>
      </div>
      <div id="piepagina">
        Éste es el pie de página
      </div>
    </div>
  </body>
</html>
```

Y en el navegador podemos ver:

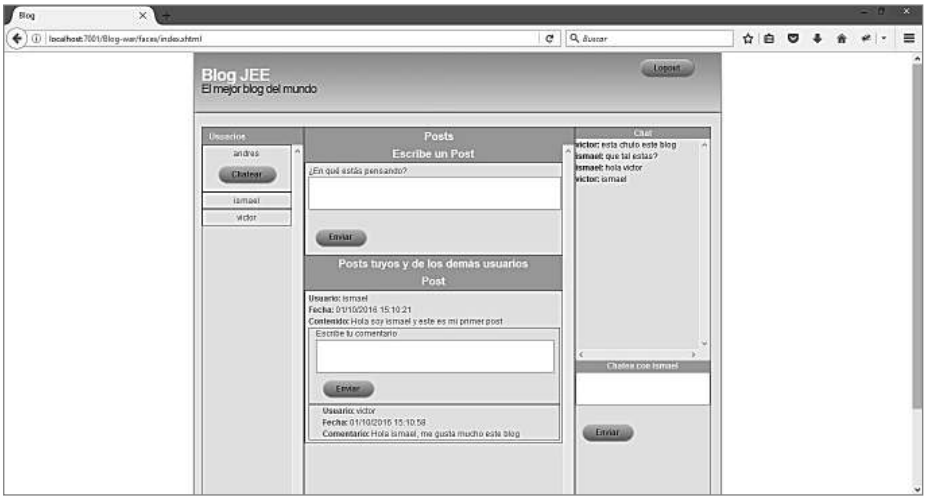
6.6 Ejercicio 5

Se propone al lector la maquetación mediante Facelets de la vista del ejercicio del blog. Tendremos una única plantilla. Mediante un *managed bean* gestionaremos el estado de dicha plantilla. El prototipo del ejercicio del blog es el siguiente:

- Vista del *login* (no se necesita registro):



- Vista del blog:



En la columna izquierda tendremos todos los usuarios que alguna vez han entrado (tengan sesión activa o no). Bajo el nombre de cada usuario tendremos un botón para indicar si en la columna derecha queremos chatear con él. En la parte central se mostrará la zona de *posts* y comentarios típica de cualquier blog. En la columna derecha aparece una zona en la que podremos mantener una conversación de chat con otro usuario registrado.

Se propone la implementación de la vista y el controlador (a falta de *websockets* para gestionar envío de *posts*, de comentarios y de chats) mediante JSF. Tendremos:

- Un *managed bean* de ámbito de sesión.
- Varios fragmentos JSF que se renderizarán o no.
  - Plantilla general: "plantillablog.hxtml".
    - "nombreusuario.xhtml". Un "div" oculto en CSS con el nombre del usuario registrado.
    - "mensajeflash.xhtml". Indicará al usuario mensajes de información o de error.
    - "cabecera.xhtml".
    - "logout.xhtml". Renderiza el botón de salida.
    - "contenidologin.xhtml".
    - "contenidoblog.xhtml".

La página "index.xhtml" heredará de "plantillablog.xhtml". Para renderizar componentes se plantea usar el componente de Facelets <ui: fragment>.

La plantilla usada utiliza una hoja de estilos propia (a desarrollar por el lector), que en el apartado de ejercicios resueltos se puede encontrar con el nombre de "style.css".

También hace uso de una hoja de estilo de botones desarrollada por Valeriu Timbuc, con licencia educacional Creative Commons y que podemos descargar desde "http://designmodo.com/futurico". Además, la plantilla es una adaptación de la que podemos encontrar en "http://www.free-css-templates.com/preview/Grey/", diseñada por Free CSS Templates con agradecimiento a Web Design US.

Para que el lector tenga una idea de qué son los elementos ocultos de nombre de usuario y de mensaje *flash*, los indicamos aquí:

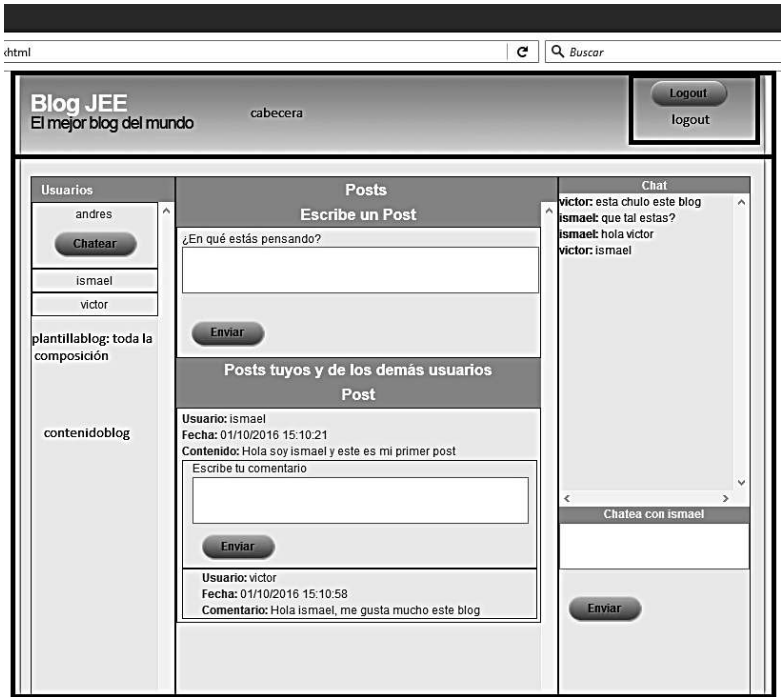
- Nombre de usuario:

```
<div id="nombreusuario">
        <!--Nombre del usuario seteado en el bean gestionado-->
</div>
```

- Mensaje *flash*:

```
<div id="mensajeflash">
  <div class="error">
    <!--Contenido del mensaje de error-->
  </div>
  <div class="mensaje">
    <!--Contenido del mensaje de información-->
  </div>
</div>
```

Y los demás elementos gráficos son los siguientes:



Se propone crear la hoja de estilos CSS, las páginas JSF y el *managed bean* de sesión para conseguir dicha estructura.

Servicios web

CAPÍTULO 7

Cuando una máquina o un conjunto de ellas ejecutan una aplicación en forma de clúster (balanceando la carga), podemos adoptar la implementación mostrada a lo largo de este manual para crear una aplicación web. Pero ¿qué ocurre con las aplicaciones distribuidas? Tenemos varias opciones:

- RMI.
- CORBA.
- Servicios web.

La solución RMI se ha estudiado someramente a lo largo de este manual y es buena y eficiente. Pero debemos importar en nuestro código las librerías de interfaces proporcionadas por el implementador de los métodos.

CORBA se plantea como una solución muy buena, ya que nos permite intercomunicar sistemas que se encuentran escritos en múltiples lenguajes de programación y corriendo sobre diversas plataformas. El "problema" es que debemos añadir una capa extra a nuestra aplicación para traducir nuestros objetos a la arquitectura de CORBA, capa extra que a lo largo de los años no se ha visto carente de problemas a la hora de su implementación. Además, si lo que queremos es enviar datos a aplicaciones cliente o a aplicaciones escritas en nuestro mismo lenguaje, CORBA se plantea innecesario (es matar una mosca a cañozanos).

Los servicios web son la solución idónea para las implementaciones en las que queremos enviar datos a clientes. Además, también permiten que los clientes estén escritos en diversos lenguajes de programación. Los clientes sólo tienen que adaptarse al protocolo del servidor.

7.1 RESTful vs. SOAP

Tradicionalmente, existen dos implementaciones para los servicios web:

- SOAP.
- RESTful.

SOAP es el acrónimo de *Simple Object Access Protocol* (Protocolo Simple de Acceso a Objetos). Los servicios SOAP son servicios web implementables en JEE mediante la librería JAX-WS. En SOAP usamos un conjunto de tecnologías para crear nuestros servicios:

- **SOAP.** Es el protocolo en sí. Necesita envolver los mensajes (los datos que enviamos) en formato SOAP. Los clientes deben adaptarse al protocolo SOAP.
- **WSDL.** Son las siglas de *Web Service Description Language* (Lenguaje de Descripción de Servicios Web). El servidor pone a disposición de los clientes un contrato en el que queda definido el servicio web en sí y la forma de acceder a él.
- **UDDI.** Es un repositorio de servicios web en forma de "diccionario". Los clientes pueden localizar los servicios web en el catálogo de registros. UDDI son las siglas de *Universal Description, Discovery and Integration* (Integración, Descubrimiento y Descripción Universal).

En el otro lado de la balanza de los servicios web tenemos los servicios RESTful. En la JEE los podemos implementar usando la API JAX-RS. Dichos servicios consisten en la creación de una API siguiendo los principios REST, que es el acrónimo de *Representational State Transfer* (Transferencia de Estado Representacional). El término fue acuñado en el año 2000 por Roy Fielding, uno de los autores de la especificación HTTP. La idea de REST es crear un canal entre cliente y servidor con las siguientes características:

- Protocolo de comunicación sin estado. Sirviéndose de forma común de HTTP.
- Una URI para cada recurso.
- Operaciones bien definidas sobre los recursos, básicamente las operaciones del CRUD *(Create, Read, Update and Delete)*.
- Sintaxis universal sin necesidad de documentación.
- Uso del principio HATEOAS. HATEOAS es el acrónimo de *Hypermedia As The Engine Of Application State* (hipermedia como motor del estado de la aplicación). Esto quiere decir que en los propios mensajes enviaremos información sobre cómo podemos seguir accediendo al servicio web.

En nuestro desarrollo, ¿cuál usaremos, SOAP o RESTful? La respuesta es RESTful, debido a:

- No requiere de una capa de código adicional, como la definición del servicio web (WSDL).

- No necesitamos adaptarnos al protocolo SOAP.
- Es más liviano para los dispositivos móviles, que por regla general serán los clientes del *back-end* de nuestra aplicación.
- Está siendo adoptado por la comunidad internacional como el estándar *de facto* para los servicios web. Facebook, Twitter, Google... han adoptado el estándar RESTful para sus servicios web.

7.2 Servicios web RESTful. JAX-RS y Jersey

Los servicios web RESTful no son más que la implementación de servicios web siguiendo los principios de diseño REST. Nosotros usaremos la librería JAX-RS y su implementación Jersey.

7.2.1 Métodos HTTP

HTTP es un protocolo sin estado. Esto quiere decir que los objetos de las operaciones del protocolo no guardan el estado conversacional del cliente. El funcionamiento de HTTP es simple: realizamos una petición *(request)*, es procesada por el servidor y se devuelve una respuesta *(response)* al cliente. Pero en la operación en sí no hay información del estado, sino que es en el cliente y en el servidor donde se guarda la información del estado de la conversación. En el cliente se guardan en las conocidas *cookies* y en el servidor se guardan en los *beans* de sesión que venimos estudiando a lo largo de este texto.

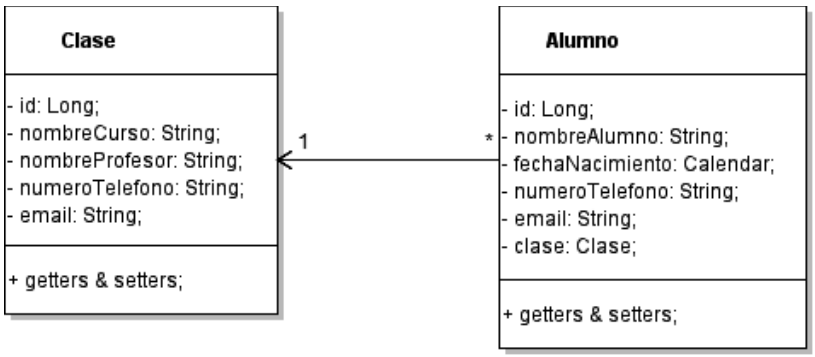
Ya hemos visto que el cliente envía una petición y el servidor devuelve una respuesta al cliente. Pero ¿cómo se sabe la operación concreta que se debe realizar? Esta operación se conoce mediante las operaciones que permite el protocolo HTTP. Éstas son:

- **GET.** Para obtener información del recurso al cual estemos accediendo en el servidor.
- **POST.** Con esta operación enviamos una nueva entrada del recurso. Es decir, con POST creamos nuevas entradas de datos.
- **PUT.** Esta operación del método HTTP nos permite actualizar el estado del recurso concreto.
- **DELETE.** Operación que nos permite eliminar un recurso del *back-end* de la aplicación.

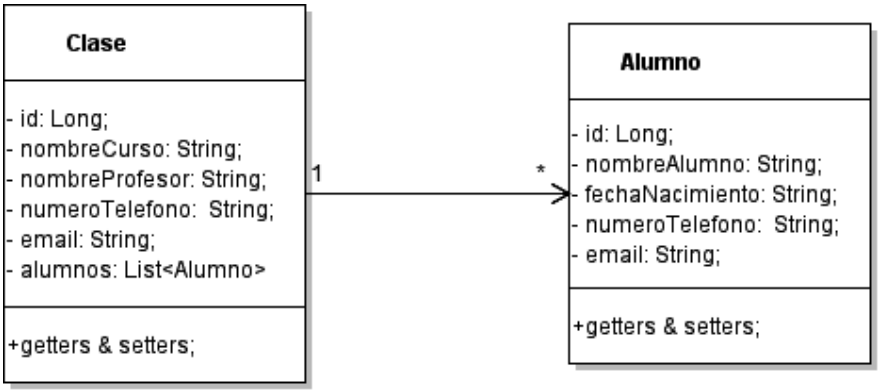
La petición y la respuesta contendrán datos, ya sea en la cabecera o en el cuerpo. Serán objetos HTTP, en los que el cuerpo será cualquier formato de datos. Para nosotros los datos viajarán en formato JSON *(JavaScript Object Notation)*.

7.2.2 Diseño de la API

Imaginemos un diagrama ERD (datos) simple:



Lo que nos daría lugar a un diagrama de clases tal cual mostramos:



Lo que a nivel de datos sería una clave ajena a nivel de dominio se convierte en un listado. Por lo tanto, podemos diseñar una API RESTful de la siguiente forma:

- Obtener los datos de todas las clases (en el caso de que hubiera varias).
  - GET: "/clase".
- Obtener los datos de una clase concreta (en el caso de que hubiera varias).
  - GET: "/clase/{id}".
- Crear un recurso de tipo "Clase".
  - POST: "/clase". En la *request* enviamos los datos de la clase.
- Actualizar un recurso de tipo "Clase".
  - PUT: "/clase/{id}". En la *request* enviamos los nuevos datos. El "id" no puede cambiar porque es el campo clave.

- Borrar un recurso de tipo "Clase".
  - DELETE: "/clase/{id}". Eliminaría la clase. Si la política de eliminación es en cascada, también eliminar los alumnos de dicha clase.
- Obtener todos los alumnos de una clase.
  - GET: "/clase/{id}/alumno".
- Obtener los datos de un alumno concreto.
  - GET: "/clase/{id}/alumno/{id}".
- Crear un recurso de tipo alumno.
  - POST: "/clase/{id}/alumno". Enviamos los datos del alumno en el cuerpo de la *request*.
- Actualizar un recurso de tipo "Alumno".
  - PUT: "/clase/{id}/alumno/{id}". En la *request* enviamos los nuevos datos del alumno. El "id" no puede cambiar porque es el campo clave.
- Borrar un recurso de tipo "Alumno".
  - DELETE: "/clase/{id}/alumno/{id}". Eliminaría al alumno concreto.

Éste sería el diseño de nuestra API RESTful. Para ser RESTful pura debemos incluir enlaces en las respuestas, siguiendo el principio HATEOAS. Imaginemos que realizamos la siguiente *request*:

- GET: "/clase/1".

Con dicha petición estamos solicitando los datos de la clase cuyo "id" es 1. En el cuerpo de la respuesta (JSON) podríamos tener:

```
{
        "id": 1,
        "nombreCurso": "Curso de Patinaje",
        "nombreProfesor" : "Ismael López Quintero",
        "numeroTelefono" : "+34. 111 11 11 11",
        "email" : "info@cursodepatinaje.com",
        "links" : [
                {
                        "rel" : "alumnos",
                        "href" : "/clase/1/alumno"
                }
        ]
}
```

El principio HATEOAS se cumple al incluir información de los enlaces a las clases referenciadas en el objeto JSON que devolvemos como respuesta.

7.2.3 Códigos de estado HTTP

Los códigos de estado en HTTP son valores numéricos devueltos por el servidor en la cabecera de la respuesta que sirven para indicar el estado del recién realizado proceso. Estos valores no pertenecen a la definición de servicios web, sino a la definición del protocolo HTTP en sí. Tenemos varios tipos de códigos (las X son cualquier dígito):

- 1XX: Información del servidor.
- 2XX: Éxito en la operación.
- 3XX: Redirecciones en la petición.
- 4XX: Error en el cliente.
- 5XX: Error en el servidor.

Los códigos que enviaremos en nuestras respuestas serán los siguientes:

Operación	Método	Código Estado
Obtener	GET	• Éxito: 200. • No encontrado: 404. • Error servidor: 500.
Crear	POST	• Éxito creado: 200 o 201. • Mala petición: 400. • Tipo no soportado: 415. • Error servidor: 500.
Borrar	DELETE	• Éxito borrado: 200 o 204. • No encontrado: 404. • Error servidor: 500.
Editar	PUT	• Éxito: 200. • Mala petición: 400. • No encontrado: 404. • Tipo no soportado: 415. • Error servidor: 500.

Con el código en la cabecera de la respuesta seremos capaces de averiguar qué ha ocurrido durante el proceso.

7.2.4 Desarrollo del *back-end*

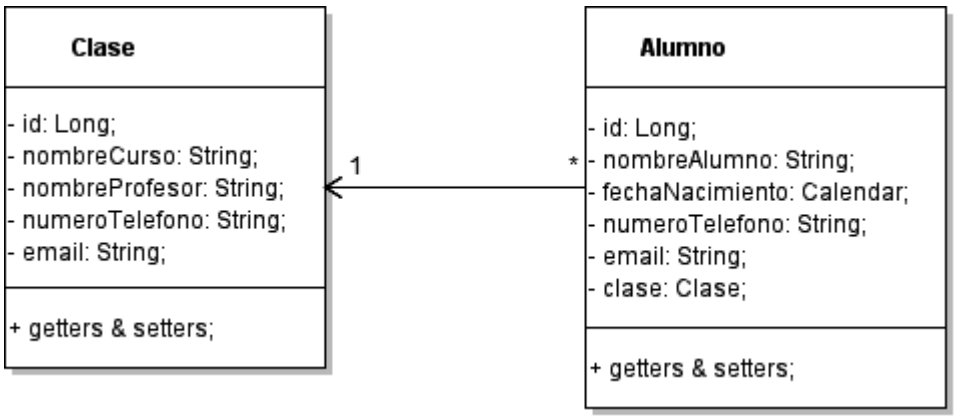
Lo primero que debemos hacer es definir la base de datos. Creamos una base de datos en MySQL llamada *cursos*, y accederemos a ella desde el servidor de aplicaciones. Acto seguido creamos un proyecto que desplegamos sobre nuestro servidor de aplicaciones. El proyecto se llama *Cursos* con sus respectivos módulos EJB y WAR. En el módulo

EJB definimos la unidad de persistencia, las entidades y los *stateless session beans* para dichas entidades (fachadas).

- Unidad de persistencia "persistence.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
 <persistence-unit name="Cursos-ejbPU" transaction-type="JTA">
  <jta-data-source>cursos</jta-data-source>
  <exclude-unlisted-classes>false</exclude-unlisted-classes>
  <properties>
   <property name="javax.persistence.schema-generation.database.action"
         value="create-or-extend-tables"/>
  </properties>
 </persistence-unit>
</persistence>
```

Las entidades se corresponden con el modelo que hemos visto anteriormente:



- Entidad "Clase.java":

```
package entities;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```

```
@Entity
public class Clase implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String nombreCurso;
   private String nombreProfesor;
   private String numeroTelefono;
   private String email;
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public String getNombreCurso() {
      return nombreCurso;
   }
   public void setNombreCurso(String nombreCurso) {
      this.nombreCurso = nombreCurso;
   }
   public String getNombreProfesor() {
      return nombreProfesor;
   }
   public void setNombreProfesor(String nombreProfesor) {
      this.nombreProfesor = nombreProfesor;
   }
   public String getNumeroTelefono() {
      return numeroTelefono;
   }
   public void setNumeroTelefono(String numeroTelefono) {
      this.numeroTelefono = numeroTelefono;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }
   @Override
   public int hashCode() {
      int hash = 0;
      hash += (id != null ? id.hashCode() : 0);
      return hash;
   }
```

```
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Clase)) {
            return false;
        }
        Clase other = (Clase) object;
        if ((this.id == null && other.id != null) ||
                        (this.id != null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "entities.Clase[ id=" + id + " ]";
    }
}
```

- Entidad "Alumno.java":

```
package entities;
import java.io.Serializable;
import java.util.Calendar;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
@Entity
public class Alumno implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nombreAlumno;
    private Calendar fechaNacimiento;
    private String numeroTelefono;
    private String email;
    @ManyToOne
    private Clase clase;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
```

```
public String getNombreAlumno() {
   return nombreAlumno;
}
public void setNombreAlumno(String nombreAlumno) {
   this.nombreAlumno = nombreAlumno;
}
public Calendar getFechaNacimiento() {
   return fechaNacimiento;
}
public void setFechaNacimiento(Calendar fechaNacimiento) {
   this.fechaNacimiento = fechaNacimiento;
}
public String getNumeroTelefono() {
   return numeroTelefono;
}
public void setNumeroTelefono(String numeroTelefono) {
   this.numeroTelefono = numeroTelefono;
}
public String getEmail() {
   return email;
}
public void setEmail(String email) {
   this.email = email;
}

public Clase getClase() {
   return clase;
}
public void setClase(Clase clase) {
   this.clase = clase;
}
@Override
public int hashCode() {
   int hash = 0;
   hash += (id != null ? id.hashCode() : 0);
   return hash;
}
@Override
public boolean equals(Object object) {
   if (!(object instanceof Alumno)) {
      return false;
   }
   Alumno other = (Alumno) object;
   if ((this.id == null && other.id != null) ||
         (this.id != null && !this.id.equals(other.id))) {
      return false;
   }
   return true;
```

```
    }
    @Override
    public String toString() {
        return "entities.Alumno[ id=" + id + " ]";
    }
}
```

Y del mismo modo definimos las fachadas en el paquete "entitiesfacades".

- Clase "AbstractFacade.java". Implementa la funcionalidad. La misma que hemos estudiado a lo largo del texto:

```
package entitiesfacades;
import java.util.List;
import javax.persistence.EntityManager;
public abstract class AbstractFacade<T> {
    private Class<T> entityClass;
    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }
    protected abstract EntityManager getEntityManager();
    public void create(T entity) {
        getEntityManager().persist(entity);
    }
    public void edit(T entity) {
        getEntityManager().merge(entity);
    }
    public void remove(T entity) {
        getEntityManager().remove(getEntityManager().merge(entity));
    }
    public T find(Object id) {
        return getEntityManager().find(entityClass,(Long) id);
    }
    public List<T> findAll() {
        javax.persistence.criteria.CriteriaQuery cq =
                getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        return getEntityManager().createQuery(cq).getResultList();
    }
    public List<T> findRange(int[] range) {
        javax.persistence.criteria.CriteriaQuery cq =
                getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        q.setMaxResults(range[1] - range[0] + 1);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }
    public int count() {
        javax.persistence.criteria.CriteriaQuery cq =
```

```
            getEntityManager().getCriteriaBuilder().createQuery();
        javax.persistence.criteria.Root<T> rt = cq.from(entityClass);
        cq.select(getEntityManager().getCriteriaBuilder().count(rt));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }
}
```

- Interfaz local de clase "ClaseFacadeLocal.java":

```
package entitiesfacades;
import entities.Clase;
import java.util.List;
import javax.ejb.Local;
@Local
public interface ClaseFacadeLocal {
    void create(Clase clase);
    void edit(Clase clase);
    void remove(Clase clase);
    Clase find(Object id);
    List<Clase> findAll();
    List<Clase> findRange(int[] range);
    int count();
}
```

- Interfaz local de alumno "AlumnoFacadeLocal.java":

```
package entitiesfacades;
import entities.Alumno;
import java.util.List;
import javax.ejb.Local;
@Local
public interface AlumnoFacadeLocal {
    void create(Alumno alumno);
    void edit(Alumno alumno);
    void remove(Alumno alumno);
    Alumno find(Object id);
    List<Alumno> findAll();
    List<Alumno> findRange(int[] range);
    int count();
    // Añadido.

    public List<Alumno> getAlumnosByClaseId(Long claseId);
}
```

Observamos el método "getAlumnosByClaseId" añadido. A continuación, la implementación de las interfaces (prácticamente toda la funcionalidad está en "AbstractFacade.java"):

- "ClaseFacade.java":

```
package entitiesfacades;
import entities.Clase;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Stateless
public class ClaseFacade extends AbstractFacade<Clase> implements ClaseFacadeLocal {
   @PersistenceContext(unitName = "Cursos-ejbPU")
   private EntityManager em;
   @Override
   protected EntityManager getEntityManager() {
      return em;
   }
   public ClaseFacade() {
      super(Clase.class);
   }
}
```

- "AlumnoFacade.java":

```
package entitiesfacades;
import entities.Alumno;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
@Stateless
public class AlumnoFacade extends AbstractFacade<Alumno>
      implements AlumnoFacadeLocal {
   @PersistenceContext(unitName = "Cursos-ejbPU")
   private EntityManager em;
   @Override
   protected EntityManager getEntityManager() {
      return em;
   }
   public AlumnoFacade() {
      super(Alumno.class);
   }
   @Override
   public List<Alumno> getAlumnosByClaseId(Long claseId) {
      Query queryString  = this.getEntityManager().createQuery("SELECT a FROM Alumno "
            + "a WHERE a.clase.id = \""+claseId+"\" ",entities.Alumno.class);
      List<Alumno> alumnos = queryString.getResultList();
      return alumnos;
   }
}
```

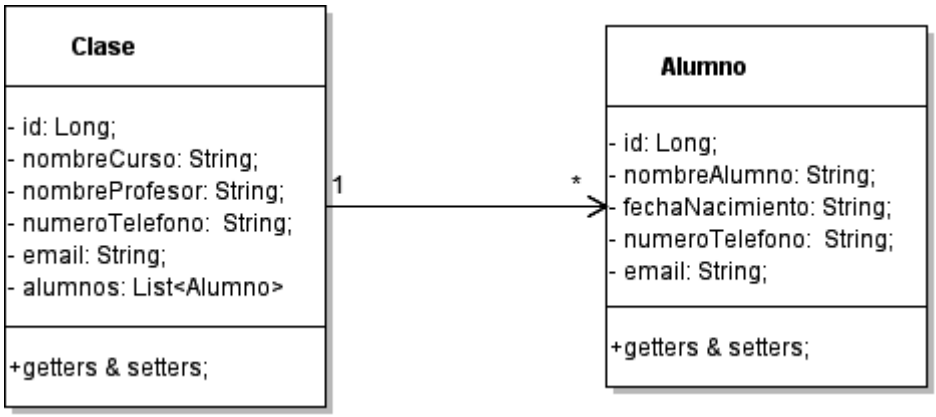
Tenemos métodos de acceso a la base de datos. Ahora, veamos cómo implementar la API.

7.2.5 CRUD con un recurso

Llamamos *recurso* o *recurso de nivel 1* a aquel en el que no necesitamos identificar recursos de orden superior para poder acceder a ellos. En nuestro caso el recurso de orden 1 es la clase. El subrecurso sería el alumno porque en nuestra lógica de negocio hemos decidido filtrar los alumnos por clase. Si queremos definir las operaciones del CRUD para el recurso, debemos definir las siguientes operaciones de nuestra API:

- Obtener los datos de todas las clases (en el caso de que hubiera varias).
  - GET: "/clase".
- Obtener los datos de una clase concreta (en el caso de que hubiera varias).
  - GET: "/clase/{id}".
- Crear un recurso de tipo "Clase".
  - POST: "/clase". En la *request* enviamos los datos de la clase.
- Actualizar un recurso de tipo "Clase".
  - PUT: "/clase/{id}". En la *request* enviamos los nuevos datos. El "id" no puede cambiar porque es el campo clave.
- Borrar un recurso de tipo "Clase".
  - DELETE: "/clase/{id}". Eliminaría la clase. Si la política de eliminación es en cascada, también eliminar los alumnos de dicha clase.

Para ello, vamos a definir en el módulo web nuestros objetos del dominio que serán los que se traduzcan a objetos hipermedia (XML o JSON). Básicamente, lo que haremos será eliminar las claves ajenas. El diagrama es el que vimos en un apartado previo.



Esto es, cada "Clase" contendrá un listado de alumnos, pero dicho listado no lo plasmaremos a nivel de *bean*, sino a nivel de subrecurso. Veamos cómo quedan de-

finidos nuestros *beans* del dominio en el módulo WAR. Les anteponemos la letra "D" como nota para indicar que son entidades del dominio, en concreto del acceso a datos.

- Clase "DClase.java":

```
package domain;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement
public class DClase implements Serializable {
   private Long id;
   private String nombreCurso;
   private String nombreProfesor;
   private String numeroTelefono;
   private String email;
   private List<Enlace> enlaces;
   public DClase() {
      this.enlaces = new ArrayList<Enlace>();
   }
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public String getNombreCurso() {
      return nombreCurso;
   }
   public void setNombreCurso(String nombreCurso) {
      this.nombreCurso = nombreCurso;
   }
   public String getNombreProfesor() {
      return nombreProfesor;
   }
   public void setNombreProfesor(String nombreProfesor) {
      this.nombreProfesor = nombreProfesor;
   }
   public String getNumeroTelefono() {
      return numeroTelefono;
   }
   public void setNumeroTelefono(String numeroTelefono) {
      this.numeroTelefono = numeroTelefono;
   }
   public String getEmail() {
      return email;
   }
```

```
    public void setEmail(String email) {
        this.email = email;
    }
    public List<Enlace> getEnlaces() {
        return enlaces;
    }
    public void setEnlaces(List<Enlace> enlaces) {
        this.enlaces = enlaces;
    }
    public void anadirEnlace(String link, String rel) {
        Enlace esteEnlace = new Enlace(link,rel);
        this.enlaces.add(esteEnlace);
    }
}
```

- Clase "DAlumno.java":

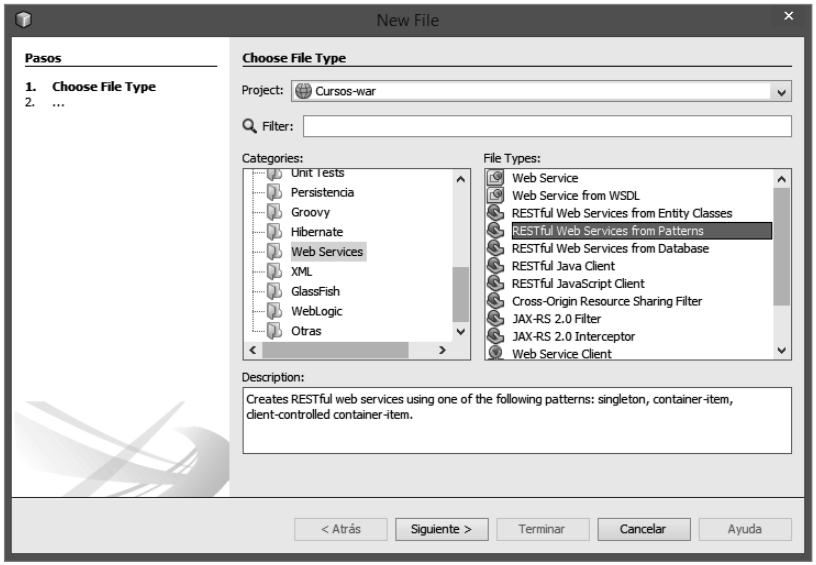
```
package domain;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement
public class DAlumno implements Serializable {
    private Long id;
    private String nombreAlumno;
    private String fechaNacimiento;
    private String numeroTelefono;
    private String email;
    private List<Enlace> enlaces;
    public DAlumno(){
        this.enlaces = new ArrayList<Enlace>();
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getNombreAlumno() {
        return nombreAlumno;
    }
    public void setNombreAlumno(String nombreAlumno) {
        this.nombreAlumno = nombreAlumno;
    }
    public String getFechaNacimiento() {
        return fechaNacimiento;
    }
    public void setFechaNacimiento(String fechaNacimiento) {
```

```
        this.fechaNacimiento = fechaNacimiento;
    }
    public String getNumeroTelefono() {
        return numeroTelefono;
    }
    public void setNumeroTelefono(String numeroTelefono) {
        this.numeroTelefono = numeroTelefono;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public List<Enlace> getEnlaces() {
        return enlaces;
    }
    public void setEnlaces(List<Enlace> enlaces) {
        this.enlaces = enlaces;
    }
    public void anadirEnlace(String link, String rel) {
        Enlace esteEnlace = new Enlace(link,rel);
        this.enlaces.add(esteEnlace);
    }
}
```

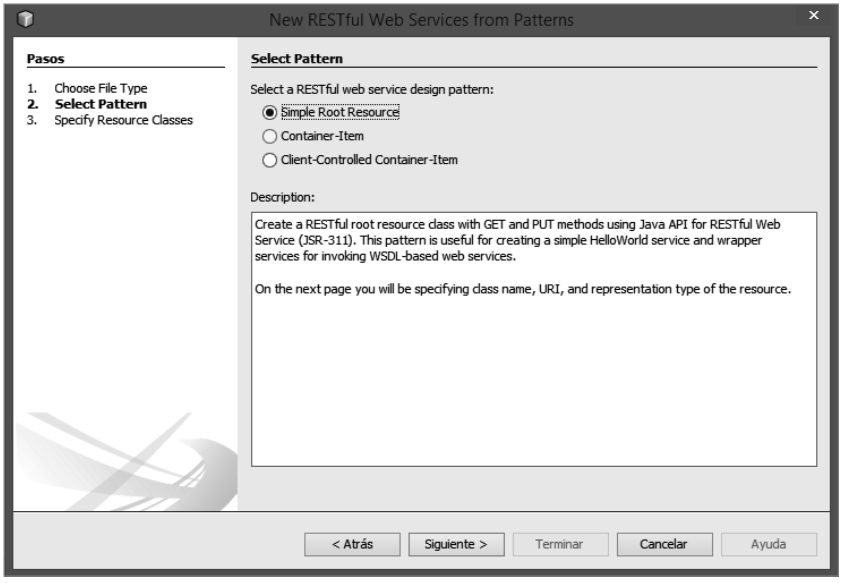
La clase "Enlace" queda implementada de la siguiente forma:

```
package domain;
public class Enlace {
    private String link;
    private String rel;
    public Enlace() {}
    public Enlace(String link, String rel) {
        this.link = link;
        this.rel = rel;
    }
    public String getLink() {
        return link;
    }
    public void setLink(String link) {
        this.link = link;
    }
    public String getRel() {
        return rel;
    }
    public void setRel(String rel) {
        this.rel = rel;
    }
}
```

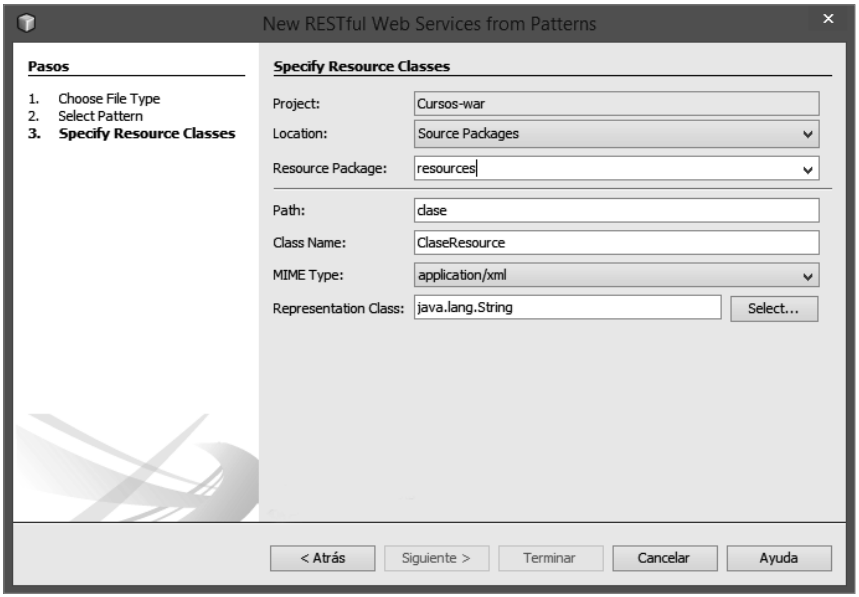
Llega el momento de crear nuestro servicio web. Debemos añadir al proyecto WAR un nuevo servicio web siguiendo estos pasos. Creamos "RESTful Web Services from Patterns":



Elegimos "Simple Root Resource":



Y escribimos el nombre de la clase, en este caso "ClaseResource". La ruta desde la que accederemos será "/clase".



A la hora de crear el servicio web debemos saber lo siguiente:

- La forma de indicar que una clase es un recurso de un servicio web es mediante la anotación "@Path", que contiene la ruta desde la cual es accesible el recurso.
- Cada método consume y produce tipos de datos. A cada método hay que indicarle el tipo que produce y consume, o bien indicarlo de manera global a toda la clase. En nuestro caso lo indicamos de manera global. Lo realizamos con anotaciones.
- La variable que seteemos con "@Context" contendrá el contexto de la petición al servicio y podremos obtener de ella información de la petición.
- A cada método le antepondremos una notación con el método HTTP con el que accederemos a él. Del mismo modo, a cada método le podremos ampliar la ruta global para toda la clase.
- Haremos uso de los códigos de estado vistos para devolver, mediante el manejo de excepciones, el código completo.

Al realizar los pasos anteriores se habrá creado una clase de configuración de nuestros servicios web. Tal clase se llama "ApplicationConfig.java" y su código queda tal cual sigue:

```
package resources;
import java.util.Set;
import javax.ws.rs.core.Application;
@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {
```

```
    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        try {
            Class jacksonProvider =
                    Class.forName("org.codehaus.jackson.jaxrs.JacksonJsonProvider");
            resources.add(jacksonProvider);
        } catch (ClassNotFoundException ex) {
            java.util.logging.Logger.getLogger(getClass().getName()).log
                (java.util.logging.Level.SEVERE, null, ex);
        }
        addRestResourceClasses(resources);
        return resources;
    }
    private void addRestResourceClasses(Set<Class<?>> resources) {
        resources.add(resources.ClaseResource.class);
    }
}
```

Sin más, pasamos a ver cómo quedaría la API que hemos diseñado para un recurso (el recurso "/clase/").

```
package resources;
import domain.DClase;
import entities.Clase;
import entitiesfacades.ClaseFacadeLocal;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import java.util.List;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Produces;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response.Status;

@Path("/clase")
@Consumes(MediaType.APPLICATION_JSON)
```

```
@Produces(MediaType.APPLICATION_JSON)
public class ClaseResource {
   ClaseFacadeLocal claseFacade = lookupClaseFacadeLocal();
   @Context
   private UriInfo context;
   public ClaseResource() {}
   @GET
   public List<DClase> getClases() {
      List<DClase> dClases = new ArrayList<>();
      try {
         List<Clase> clases = claseFacade.findAll();
         for (Clase clase : clases) {
            DClase nuevaDClase = this.convertClaseFromDBToDomain(clase);
            nuevaDClase = this.anadirEnlaces(nuevaDClase);
            dClases.add(nuevaDClase);
         }
      } catch (Exception e) {
         throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
      }
      return dClases;
   }
   @POST
   public DClase crearClase(DClase dClase) {
      if (dClase == null) {
         throw new WebApplicationException(Status.BAD_REQUEST);
      }
      try {
         Clase clase = null;
         clase = this.convertClaseFromDomainToDB(dClase);
         claseFacade.create(clase);
         dClase = this.convertClaseFromDBToDomain(clase);
         dClase = this.anadirEnlaces(dClase);
      } catch (Exception ex) {
         throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
      }
      return dClase;
   }
   @GET
   @Path("/{claseId}")
   public DClase getClaseById(@PathParam("claseId") String claseId) {
      Long id = null;
      if ((claseId == null) || (claseId.trim()).equals("")) {
         throw new WebApplicationException(Status.BAD_REQUEST);
      }
      try {
         id = Long.valueOf(claseId);
```

```
        } catch (Exception e) {
            throw new WebApplicationException(Status.BAD_REQUEST);
        }
        Clase clase = null;
        DClase dClase = null;
        clase = claseFacade.find(id);
        if (clase == null) {
            throw new WebApplicationException(Status.NOT_FOUND);
        }
        try {
            dClase = this.convertClaseFromDBToDomain(clase);
            dClase = this.anadirEnlacesConId(dClase);
        } catch (Exception ex) {
            throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
        }
        return dClase;
    }
    @PUT
    @Path("/{claseId}")
    public DClase updateClaseById(@PathParam("claseId") String claseId, DClase dClase) {
        Long id = null;
        if ((claseId == null) || (claseId.trim()).equals("")) {
            throw new WebApplicationException(Status.BAD_REQUEST);
        }
        try {
            id = Long.valueOf(claseId);
        } catch (Exception e) {
            throw new WebApplicationException(Status.BAD_REQUEST);
        }
        if (dClase == null) {
            throw new WebApplicationException(Status.BAD_REQUEST);
        }
        Clase clase = null;
        Clase claseEnBD = null;
        claseEnBD = claseFacade.find(id);
        if (claseEnBD == null) {
            throw new WebApplicationException(Status.NOT_FOUND);
        }
        try {
            clase = this.convertClaseFromDomainToDB(dClase);
            clase.setId(Long.valueOf(claseId));
            claseFacade.edit(clase);
            dClase = this.convertClaseFromDBToDomain(clase);
            dClase = this.anadirEnlacesConId(dClase);
        } catch (Exception e) {
            throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
        }
        return dClase;
    }
```

```
@DELETE
@Path("/{claseId}")
public DClase removeClaseById(@PathParam("claseId") String claseId) {
   Long id = null;
   if ((claseId == null) || (claseId.trim()).equals("")) {
      throw new WebApplicationException(Status.BAD_REQUEST);
   }
   try {
      id = Long.valueOf(claseId);
   } catch (Exception e) {
      throw new WebApplicationException(Status.BAD_REQUEST);
   }
   Clase clase = null;
   DClase dClase = null;
   clase = claseFacade.find(id);
   if (clase == null) {
      throw new WebApplicationException(Status.NOT_FOUND);
   }
   try {
      dClase = this.convertClaseFromDBToDomain(clase);
      dClase = this.anadirEnlacesConId(dClase);
      claseFacade.remove(clase);
   } catch (Exception e) {
      throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
   }
   return dClase;
}
@Path("/{claseId}/alumno")
public AlumnoResource getAlumnoResource() {
   return new AlumnoResource();
}
private ClaseFacadeLocal lookupClaseFacadeLocal() {
   try {
      javax.naming.Context c = new InitialContext();
      return (ClaseFacadeLocal) c.lookup("java:global/Cursos/Cursos-ejb/"
            + "ClaseFacade!entitiesfacades.ClaseFacadeLocal");
   } catch (NamingException ne) {
      System.out.println(ne.getMessage());
      throw new RuntimeException(ne);
   }
}
private DClase convertClaseFromDBToDomain(Clase clase) {
   DClase dClase = new DClase();
   dClase.setId(clase.getId());
   dClase.setNombreCurso(clase.getNombreCurso());
   dClase.setNombreProfesor(clase.getNombreProfesor());
   dClase.setNumeroTelefono(clase.getNumeroTelefono());
   dClase.setEmail(clase.getEmail());
   return dClase;
}
```

```
    private Clase convertClaseFromDomainToDB(DClase dClase)
            throws UnsupportedEncodingException {
        Clase clase = new Clase();
        clase.setId(dClase.getId());
        clase.setNombreCurso(new String(dClase.getNombreCurso()
                .getBytes("ISO-8859-1"), "UTF-8"));
        clase.setNombreProfesor(new String(dClase.getNombreProfesor()
                .getBytes("ISO-8859-1"), "UTF-8"));
        clase.setNumeroTelefono(new String(dClase.getNumeroTelefono()
                .getBytes("ISO-8859-1"), "UTF-8"));
        clase.setEmail(new String(dClase.getEmail()
                .getBytes("ISO-8859-1"), "UTF-8"));
        return clase;
    }
    private DClase anadirEnlaces(DClase dClase) {
        Long claseId = dClase.getId();
        String rel = "alumnos";
        String link = context.getAbsolutePathBuilder()
                .path(claseId.toString())
                .path("/alumno")
                .build().toString();
        dClase.anadirEnlace(link, rel);
        return dClase;
    }
    private DClase anadirEnlacesConId(DClase dClase) {
        Long claseId = dClase.getId();
        String rel = "alumnos";
        String link = context.getAbsolutePathBuilder()
                .path("/alumno")
                .build().toString();
        dClase.anadirEnlace(link, rel);
        return dClase;
    }
}
```

Como ejemplo de funcionamiento, realicemos un conjunto de llamadas sobre la API con el recurso desarrollado. Usaremos el cliente "Postman", que forma parte de Google WebStore (de esta forma podemos testar nuestra API sin necesidad de programar un cliente).

- Petición GET.
  - URL: "http://localhost:7001/Cursos-war/webresources/clase".
  - Datos: (vacío).
  - Respuesta: [].

- Petición POST.
  - URL: "http://localhost:7001/Cursos-war/webresources/clase".
  - Datos:

```
{
  "email": "info@cursodepatinaje.com",
  "nombreCurso": "Curso de Patinaje",
  "nombreProfesor": "Javier Hernández Rodríguez",
  "numeroTelefono": "+34. 111 11 11 11"
}
```

  - Respuesta:

```
{
 "email": "info@cursodepatinaje.com",
 "enlaces": [
  {
   "link": "http://localhost:7001/Cursos-war/webresources/clase/1/alumno",
    "rel": "alumnos"
  }
 ],
 "id": 1,
 "nombreCurso": "Curso de Patinaje",
 "nombreProfesor": "Javier Hernández Rodríguez",
 "numeroTelefono": "+34. 111 11 11 11"
}
```

Añadimos, del mismo modo, mediante operación POST los siguientes cursos a nuestra base de datos:

- Curso:

```
{
  "email": "info@cursodeingles.com",
  "nombreCurso": "Curso de Inglés",
  "nombreProfesor": "Alberto López Méndez",
  "numeroTelefono": "+34. 222 22 22 22"
}
```

- Curso.

```
{
  "email": "info@cursodebuzo.com",
  "nombreCurso": "Curso de Buzo",
  "nombreProfesor": "Antonio Cruz Pérez",
  "numeroTelefono": "+34. 333 33 33 33"
}
```

- Petición GET.
  - URL: "http://localhost:7001/Cursos-war/webresources/clase".
  - Datos: (vacío).

- Respuesta:


```
[
 {
  "email": "info@cursodepatinaje.com",
  "enlaces": [
   {
      "link": "http://localhost:7001/Cursos-war/webresources/clase/1/alum-
no",
     "rel": "alumnos"
   }
  ],
  "id": 1,
  "nombreCurso": "Curso de Patinaje",
  "nombreProfesor": "Javier Hernández Rodríguez",
  "numeroTelefono": "+34. 111 11 11 11"
 },
 {
  "email": "info@cursodeingles.com",
  "enlaces": [
   {
      "link": "http://localhost:7001/Cursos-war/webresources/clase/2/alum-
no",
     "rel": "alumnos"
   }
  ],
  "id": 2,
  "nombreCurso": "Curso de Inglés",
  "nombreProfesor": "Alberto López Méndez",
  "numeroTelefono": "+34. 222 22 22 22"
 },
 {
  "email": "info@cursodebuzo.com",
  "enlaces": [
   {
      "link": "http://localhost:7001/Cursos-war/webresources/clase/3/alum-
no",
     "rel": "alumnos"
   }
  ],
  "id": 3,
  "nombreCurso": "Curso de Buzo",
  "nombreProfesor": "Antonio Cruz Pérez",
  "numeroTelefono": "+34. 333 33 33 33"
 }
]
```

- Petición GET.
  - URL: "http://localhost:7001/Cursos-war/webresources/clase/2".
  - Datos: (vacío).
  - Respuesta:


```
{
 "email": "info@cursodeingles.com",
 "enlaces": [
  {
   "link": "http://localhost:7001/Cursos-war/webresources/clase/2/alumno",
    "rel": "alumnos"
  }
 ],
 "id": 2,
 "nombreCurso": "Curso de Inglés",
 "nombreProfesor": "Alberto López Méndez",
 "numeroTelefono": "+34. 222 22 22 22"
}
```


- Petición PUT (para actualizar).
  - URL: "http://localhost:7001/Cursos-war/webresources/clase/1".
  - Datos:


```
{
   "email": "info@cursodepatinajeavanzado.com",
   "nombreCurso": "Curso de Patinaje Avanzado",
   "nombreProfesor": "Javier Hernández Rodríguez",
   "numeroTelefono": "+34. 111 11 11 11"
}
```


  - Respuesta:


```
{
 "email": "info@cursodepatinajeavanzado.com",
 "enlaces": [
  {
   "link": "http://localhost:7001/Cursos-war/webresources/clase/1/alumno",
    "rel": "alumnos"
  }
 ],
 "id": 1,
 "nombreCurso": "Curso de Patinaje Avanzado",
 "nombreProfesor": "Javier Hernández Rodríguez",
 "numeroTelefono": "+34. 111 11 11 11"
}
```

- Petición DELETE.
  - URL: "http://localhost:7001/Cursos-war/webresources/clase/2".
  - Datos: (vacío).
  - Respuesta:

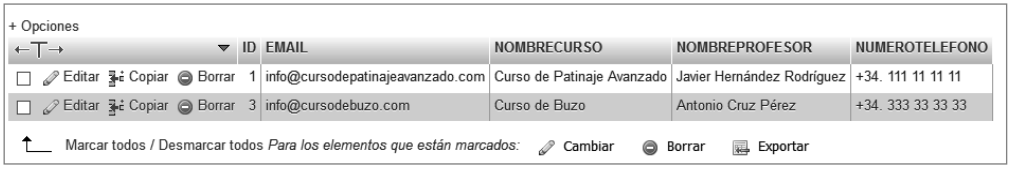

```
{
 "email": "info@cursodeingles.com",
 "enlaces": [
  {
   "link": "http://localhost:7001/Cursos-war/webresources/clase/2/alumno",
    "rel": "alumnos"
  }
 ],
 "id": 2,
 "nombreCurso": "Curso de Inglés",
 "nombreProfesor": "Alberto López Méndez",
 "numeroTelefono": "+34. 222 22 22 22"
}
```


- Petición GET (vemos como ya no está el curso de inglés y el curso de patinaje es avanzado).
  - URL: http://localhost:7001/Cursos-war/webresources/clase.
  - Datos: (vacío).
  - Respuesta:


```
[
 {
  "email": "info@cursodepatinajeavanzado.com",
  "enlaces": [
   {
    "link": "http://localhost:7001/Cursos-war/webresources/clase/1/alumno",
    "rel": "alumnos"
   }
  ],
  "id": 1,
  "nombreCurso": "Curso de Patinaje Avanzado",
  "nombreProfesor": "Javier Hernández Rodríguez",
  "numeroTelefono": "+34. 111 11 11 11"
 },
 {
  "email": "info@cursodebuzo.com",
  "enlaces": [
   {
```

```
      "link": "http://localhost:7001/Cursos-war/webresources/clase/3/alumno",
      "rel": "alumnos"
    }
  ],
  "id": 3,
  "nombreCurso": "Curso de Buzo",
  "nombreProfesor": "Antonio Cruz Pérez",
  "numeroTelefono": "+34. 333 33 33 33"
 }
]
```

Veamos el estado de nuestra tabla de cursos en el cliente MySQL:



7.2.6 CRUD con subrecursos

Un subrecurso es enlazable desde un recurso principal. El lector entenderá perfectamente el concepto de subrecurso si piensa en el alumno de nuestro ejemplo. No tiene sentido obtener todos los alumnos de todos los cursos (aunque también los podríamos obtener). Las consultas tendrán sentido si obtenemos los alumnos de un curso. En el recurso de "clase" hemos enlazado con el subrecurso de "alumno" con el siguiente método:

```
@Path("/{claseId}/alumno")
public AlumnoResource getAlumnoResource() {
  return new AlumnoResource();
}
```

De hecho, los métodos que queremos implementar ahora son:

- Obtener todos los alumnos de una clase.
  - GET: "/clase/{id}/alumno".
- Obtener los datos de un alumno concreto.
  - GET: "/clase/{id}/alumno/{id}".
- Crear un recurso de tipo "Alumno".
  - POST: "/clase/{id}/alumno". Enviamos los datos del alumno en el cuerpo de la *request*.

- Actualizar un recurso de tipo "Alumno".
  - PUT: "/clase/{id}/alumno/{id}". En la *request* enviamos los nuevos datos del alumno. El "id" no puede cambiar porque es el campo clave.
- Borrar un recurso de tipo "Alumno".
  - DELETE: /clase/{id}/alumno/{id}. Eliminaría al alumno concreto.

Para ello, nuestro nuevo recurso debe estar definido con la ruta "@Path("/")", ya que no será accesible desde la ruta principal, sino a través de la ruta de un recurso de nivel 1.

Añadimos el recurso "Alumno", y nuestra clase "ApplicationConfig" queda tal cual sigue:

```
package resources;
import java.util.Set;
import javax.ws.rs.core.Application;
@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {
   @Override
   public Set<Class<?>> getClasses() {
      Set<Class<?>> resources = new java.util.HashSet<>();
      try {
         Class jacksonProvider =
               Class.forName("org.codehaus.jackson.jaxrs.JacksonJsonProvider");
         resources.add(jacksonProvider);
      } catch (ClassNotFoundException ex) {
         java.util.logging.Logger.getLogger(getClass().getName()).log
            (java.util.logging.Level.SEVERE, null, ex);
      }
      addRestResourceClasses(resources);
      return resources;
   }
   private void addRestResourceClasses(Set<Class<?>> resources) {
      resources.add(resources.AlumnoResource.class);
      resources.add(resources.ClaseResource.class);
   }
}
```

Vemos que en el método "addRestResourceClasses" se ha añadido la línea: "resources.add(resources.AlumnoResource.class);". Veamos la implementación de la clase "AlumnoResource.java":

```
package resources;
import domain.DAlumno;
import entities.Alumno;
import entities.Clase;
import entitiesfacades.AlumnoFacadeLocal;
```

```
import entitiesfacades.ClaseFacadeLocal;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.List;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response.Status;
@Path("/")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class AlumnoResource {
   ClaseFacadeLocal claseFacade = lookupClaseFacadeLocal();
   AlumnoFacadeLocal alumnoFacade = lookupAlumnoFacadeLocal();
   @Context
   private UriInfo context;
   public AlumnoResource() {}
   @GET
   public List<DAlumno> getAlumnosByClaseId(@PathParam("claseId") String claseId) {
      Long lClaseId = null;
      List<DAlumno> dAlumnos = new ArrayList<DAlumno>();
      if ((claseId == null) || ((claseId.trim()).equals(""))) {
         throw new WebApplicationException(Status.BAD_REQUEST);
      }
      try {
         lClaseId = Long.valueOf(claseId);
      } catch(Exception e) {
         throw new WebApplicationException(Status.BAD_REQUEST);
      }
      try {
         List<Alumno> alumnos = alumnoFacade.getAlumnosByClaseId(lClaseId);
         for(Alumno a: alumnos) {
            DAlumno dA = this.convertAlumnoFromDBToDomain(a);
            dAlumnos.add(dA);
         }
```

```
    } catch(Exception e) {
      throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
    }
    return dAlumnos;
  }
  @POST
  public DAlumno crearAlumno(@PathParam("claseId") String claseId, DAlumno dAlumno) {
    Long lClaseId = null;
    if (dAlumno == null) throw new WebApplicationException(Status.BAD_REQUEST);
    if ((claseId == null) || ((claseId.trim()).equals(""))) {
      throw new WebApplicationException(Status.BAD_REQUEST);
    }
    try {
      lClaseId = Long.valueOf(claseId);
    } catch(Exception e) {
      throw new WebApplicationException(Status.BAD_REQUEST);
    }
    Clase clase = claseFacade.find(lClaseId);
    if (clase==null) {
      throw new WebApplicationException(Status.NOT_FOUND);
    }
    try {
      Alumno alumno = this.convertAlumnoFromDomainToDB(dAlumno);
      alumno.setClase(clase);
      alumnoFacade.create(alumno);
      dAlumno = this.convertAlumnoFromDBToDomain(alumno);
    } catch(Exception e) {
      throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
    }
    return dAlumno;
  }
  @GET
  @Path("/{alumnoId}")
  public DAlumno getAlumnoById(@PathParam("claseId") String claseId,
      @PathParam("alumnoId") String alumnoId) {
    Long lClaseId = null;
    Long lAlumnoId = null;
    DAlumno dA = null;
    if ((claseId == null) || ((claseId.trim()).equals("")) ||
        (alumnoId == null) || ((alumnoId.trim()).equals(""))) {
      throw new WebApplicationException(Status.BAD_REQUEST);
    }
    try {
      lClaseId = Long.valueOf(claseId);
      lAlumnoId = Long.valueOf(alumnoId);
    } catch(Exception e) {
      throw new WebApplicationException(Status.BAD_REQUEST);
    }
```

```
    Alumno a = alumnoFacade.find(lAlumnoId);
    if (a == null) throw new WebApplicationException(Status.NOT_FOUND);
    Clase c = a.getClase();
    if (c.getId().equals(lClaseId)) {
      dA = this.convertAlumnoFromDBToDomain(a);
    } else {
      throw new WebApplicationException(Status.BAD_REQUEST);
    }
    return dA;
  }
  @PUT
  @Path("/{alumnoId}")
  public DAlumno updateAlumnoById(@PathParam("claseId") String claseId,
      @PathParam("alumnoId") String alumnoId, DAlumno dAlumno) {
    Long lClaseId = null;
    Long lAlumnoId = null;
    DAlumno dA = null;
    if ((claseId == null) || ((claseId.trim()).equals("")) ||
        (alumnoId == null) || ((alumnoId.trim()).equals(""))) {
      throw new WebApplicationException(Status.BAD_REQUEST);
    }
    try {
      lClaseId = Long.valueOf(claseId);
      lAlumnoId = Long.valueOf(alumnoId);
    } catch(Exception e) {
      throw new WebApplicationException(Status.BAD_REQUEST);
    }
    if (dAlumno == null) {
      throw new WebApplicationException(Status.BAD_REQUEST);
    }
    Alumno a = alumnoFacade.find(lAlumnoId);
    if (a == null) throw new WebApplicationException(Status.NOT_FOUND);
    Clase c = a.getClase();
    if (c.getId().equals(lClaseId)) {
      try {
        Alumno alumno = this.convertAlumnoFromDomainToDB(dAlumno);
        alumno.setId(lAlumnoId);
        alumno.setClase(c);
        alumnoFacade.edit(alumno);
        dA = this.convertAlumnoFromDBToDomain(alumno);
      } catch(Exception e) {
        throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
      }
    } else {
      throw new WebApplicationException(Status.BAD_REQUEST);
    }
    return dA;
  }
```

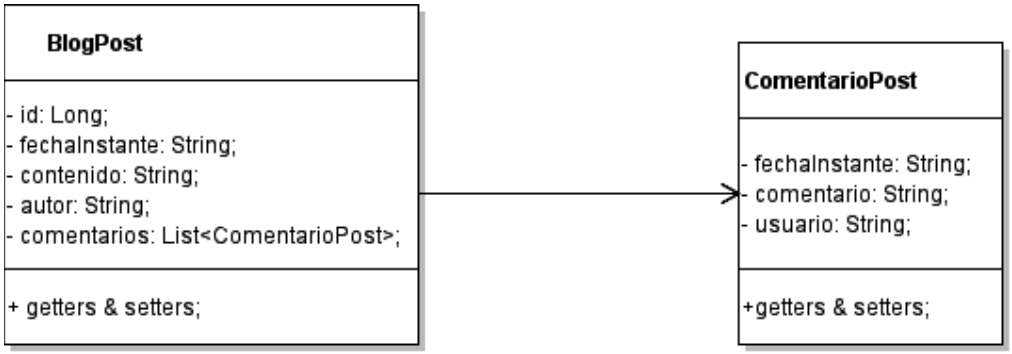
```
@DELETE
@Path("/{alumnoId}")
public DAlumno deleteAlumnoById(@PathParam("claseId") String claseId,
      @PathParam("alumnoId") String alumnoId) {
   Long lClaseId = null;
   Long lAlumnoId = null;
   DAlumno dA = null;
   if ((claseId == null) || ((claseId.trim()).equals("")) ||
         (alumnoId == null) || ((alumnoId.trim()).equals(""))) {
      throw new WebApplicationException(Status.BAD_REQUEST);
   }
   try {
      lClaseId = Long.valueOf(claseId);
      lAlumnoId = Long.valueOf(alumnoId);
   } catch(Exception e) {
      throw new WebApplicationException(Status.BAD_REQUEST);
   }
   Alumno a = alumnoFacade.find(lAlumnoId);
   if (a == null) throw new WebApplicationException(Status.NOT_FOUND);
   Clase c = a.getClase();
   if (c.getId().equals(lClaseId)) {
      dA = this.convertAlumnoFromDBToDomain(a);
      alumnoFacade.remove(a);
   } else {
      throw new WebApplicationException(Status.BAD_REQUEST);
   }
   return dA;
}
private DAlumno convertAlumnoFromDBToDomain(Alumno alumno) {
   DAlumno dAlumno = new DAlumno();
   dAlumno.setId(alumno.getId());
   dAlumno.setNombreAlumno(alumno.getNombreAlumno());
   Calendar fecha = alumno.getFechaNacimiento();
   int ano = fecha.get(Calendar.YEAR);
   int mes = fecha.get(Calendar.MONTH) + 1;
   int dia = fecha.get(Calendar.DAY_OF_MONTH);
   String diaString = (dia < 10) ?  "0" + dia : ""+dia;
   String mesString = (mes < 10) ?  "0" + mes : ""+mes;
   String anoString = "" + ano;
   String fechaNacimiento = diaString + "/" + mesString + "/" + anoString;
   dAlumno.setFechaNacimiento(fechaNacimiento);
   dAlumno.setEmail(alumno.getEmail());
   dAlumno.setNumeroTelefono(alumno.getNumeroTelefono());
   return dAlumno;
}
private Alumno convertAlumnoFromDomainToDB(DAlumno dAlumno)
      throws Exception {
   Alumno alumno = new Alumno();
   alumno.setId(dAlumno.getId());
```

```
        alumno.setNombreAlumno(new String(dAlumno.getNombreAlumno()
                .getBytes("ISO-8859-1"), "UTF-8"));
        Calendar fechaNacimiento = this.getCalendarFromString(dAlumno.getFechaNacimiento());
        alumno.setFechaNacimiento(fechaNacimiento);
        alumno.setNumeroTelefono(new String(dAlumno.getNumeroTelefono()
                .getBytes("ISO-8859-1"), "UTF-8"));
        alumno.setEmail(new String(dAlumno.getEmail()
                .getBytes("ISO-8859-1"), "UTF-8"));
        return alumno;
    }
    private Calendar getCalendarFromString(String fecha) throws Exception {
        String[] numeros = fecha.split("/");
        if (numeros.length == 3) {
            String diaString = numeros[0];
            String mesString = numeros[1];
            String anoString = numeros[2];
            try {
                int dia = Integer.parseInt(diaString);
                int mes = Integer.parseInt(mesString) - 1;
                int ano = Integer.parseInt(anoString);
                return new GregorianCalendar(ano,mes,dia);
            } catch(Exception e) {
                throw e;
            }
        } else {
            throw new Exception();
        }
    }
    private AlumnoFacadeLocal lookupAlumnoFacadeLocal() {
        try {
            javax.naming.Context c = new InitialContext();
            return (AlumnoFacadeLocal) c.lookup("java:global/Cursos/Cursos-ejb/"
                    + "AlumnoFacade!entitiesfacades.AlumnoFacadeLocal");
        } catch (NamingException ne) {
            System.out.println(ne.getMessage());
            throw new RuntimeException(ne);
        }
    }
    private ClaseFacadeLocal lookupClaseFacadeLocal() {
        try {
            javax.naming.Context c = new InitialContext();
            return (ClaseFacadeLocal) c.lookup("java:global/Cursos/Cursos-ejb/"
                    + "ClaseFacade!entitiesfacades.ClaseFacadeLocal");
        } catch (NamingException ne) {
            System.out.println(ne.getMessage());
            throw new RuntimeException(ne);
        }
    }
}
```

Con este código queda definido el servicio web completo de clases y alumnos.

7.3 Ejercicio 6

Se plantea al lector la implementación de una API RESTful usando JAX-RS para el ejercicio del blog, en concreto para los modelos de "BlogPost" y de "ComentarioPost".



El recurso de nivel 1 es "BlogPost" y el subrecurso es "ComentarioPost". En concreto, la API debe tener los siguientes métodos:

- Obtener todos los *posts*.
  - GET: "/post".
- Obtener los datos de un *post* concreto.
  - GET: "/post/{id}".
- Crear un recurso de tipo "Post". El usuario debe existir en la base de datos. Si no existe, se debe devolver un error de tipo "Bad Request".
  - POST: "/post". En la *request* enviamos los datos del *post* en formato JSON.

Y para el subrecurso:

- Obtener todos los comentarios de un *post*.
  - GET: "/post/{id}/comentario".
- Crear un recurso de tipo "Comentario".
  - POST: "/post/{id}/comentario". En la *request* enviamos los datos del comentario en formato JSON.

No pondremos enlaces a los comentarios siguiendo el principio HATEOAS, ya que nos conlleva modificar el modelo del dominio y es un ejemplo simple.

Websockets

CAPÍTULO
8

Los *websockets* son la solución del protocolo TCP (subprotocolo WS) que permite una comunicación asíncrona entre el navegador del usuario y el servidor web que le ha despachado la página. Con los *websockets* conseguimos soluciones óptimas en tiempo real, como chats y redes sociales. No necesitamos peticiones HTTP para comunicar en segundo plano el navegador con el servidor (ni peticiones HTTP tradicionales ni vía AJAX).

8.1 Lado del cliente

JavaScript de forma nativa permite la ejecución de *websockets*. La clase "WebSocket" está disponible en el navegador y podemos crear instancias de ella con la siguiente instrucción (vemos código JavaScript en el lado del cliente):

```
var webSocket = new WebSocket("ws://localhost/DireccionServidorWebSocket ");
```

Evidentemente, hemos de sustituir *localhost* por la dirección IP concreta o por el dominio que apunte a nuestro servidor web.

Es fácil enviar mensajes de texto al servidor con el *websocket* abierto. La nomenclatura que usaremos para enviar mensajes al servidor es la siguiente:

```
var mensaje = 'mensaje de texto';
swebSocket.send(mensaje);
```

Para poder tener más potencia en el envío de información, recordamos que cualquier mensaje XML o JSON se puede "codificar" en una cadena de texto. Imaginemos que queremos enviar al servidor el siguiente mensaje:

```
var operacion = {
  definicion: "saludo",
  contenido: "Hola a todos"
};
var mensaje = JSON.stringify(operacion);
// Suponemos que ya tenemos creado el websocket.
webSocket.send(mensaje);
```

Como se puede ver, la clase JSON, que también tenemos disponible de forma nativa en JavaScript, permite "codificar" cualquier objeto JavaScript en una cadena de texto.

Es tan simple el uso de *websockets* en JavaScript que sólo nos queda por estudiar dos operaciones más: una la que se lanza (evento) cuando se recibe un mensaje desde el servidor y otra la que lanza el navegador cuando quiere cerrar el *socket*. Suponiendo que tenemos la variable "webSocket" con el *socket* abierto.

```
webSocket.onmessage = function(mensaje) {
};
```

El mensaje que recibimos también vuelve a ser una cadena de texto, pero podemos usar la función de "jQuery $.parseJson(mensaje)" para obtener el objeto JSON a partir de la cadena.

La otra operación es la que cierra el *socket*:

```
webSocket.close();
```

De todas formas, el *socket* también se puede cerrar en caso de inactividad, por eso es bueno enviar cada cierto tiempo al servidor una señal indicando que el cliente sigue vivo, por ejemplo:

```
function setKeepAlive () {
  setInterval(function () {
    var operacion = {
      // definición de la operación
    };
    var mensaje = JSON.stringify(operacion);
    webSocket.send(mensaje);
  }, 4000);
};
```

Dicho fragmento de código le envía al servidor una señal cada cuatro segundos. Con ello, conseguimos que el servidor tenga constancia de la vida del cliente.

8.2 Lado del servidor

Dependiendo del lenguaje, en el lado del servidor tendremos una implementación u otra. En Java, y formando parte de JEE, tenemos la solución "ServerEndpoint", disponible en el paquete "javax.websocket.server". Hemos visto que las comunicaciones mediante *websockets* las maneja el cliente, excepto en el caso en que el servidor considere que un cliente ha expirado por inactividad. Veamos cómo implementar una clase "ServerEndpoint" de forma que actúe como *singleton* para recibir las llamadas *websocket* de la aplicación:

```
package socket;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.websocket.*;
import javax.websocket.server.ServerEndpoint;
import socket.modelo.Operacion;
import util.JSONManager;
@ServerEndpoint("/websocketserver")
public class WebSocketServer {
   private static Map<String, Session> sesiones = new HashMap<String, Session>();
   @OnOpen
   public void abrirSocket(Session sesionUsuario) {
      sesiones.put(sesionUsuario.getId(), sesionUsuario);
   }
   @OnMessage
   public void recibeMensaje(String mensaje, Session sesionUsuario)
         throws IOException {
      Object operacion = JSONManager.generateTOfromJson(mensaje, Operacion.class);
      // Tratamos la información que nos ha llegado.
      Respuesta r = new Respuesta();
      String mensajeTexto = JSONManager.generateJson(r);
      sesionUsuario.getBasicRemote().sendText(mensajeTexto);
   }
   @OnClose
   public void cerrarSocket(Session sesionUsuario) {
      sesiones.remove(sesionUsuario.getId());
   }
}
```

Si queremos conectarnos a dicho punto de servidor desde un cliente JavaScript, suponiendo que el servidor esté alojado en *localhost*, haríamos la siguiente llamada (desde JavaScript):

```
var webSocket = new WebSocket("ws://localhost/websocketserver ");
```

Con el "Map" conseguimos tener un listado de los clientes que se encuentran conectados mediante *websocket*, para hacer una comunicación *multicast*, llegado el caso.

Hemos hecho uso de la clase "JSONManager" con métodos estáticos. Dicha clase queda codificada tal y como podremos ver a continuación. Indicar que hace uso de la librería GSon, de libre distribución (la podemos descargar desde repositorio Git). Con ella conseguimos convertir objetos de Java en objetos JSON codificados en cadenas de texto, y viceversa.

La clase "JSONManager" queda tal cual sigue:

```
package util;
import com.google.gson.Gson;
public final class JSONManager {
   private static Gson gson = new Gson();
   public static String generateJson(Object to) {
      return null == to ? "" : gson.toJson(to);
   }
   public static Object generateTOfromJson(String json, Class<?> class1) {
      return gson.fromJson(json, class1);
   }
}
```

8.3 Implementación de una sala de chat

Mostremos el código de servidor y de cliente de una sala de chat. El código de servidor es el siguiente:

```
package socket;
import java.io.IOException;
import java.io.StringWriter;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import javax.json.Json;
import javax.json.JsonObject;
import javax.json.JsonWriter;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
@ServerEndpoint("/chatsocket")
public class ChatSocket {
   private static Map<String,Session> usuarios = new HashMap<String,Session>();
   @OnOpen
```

```
  public void abrirSocket(Session sesionUsuario) {
    usuarios.put(sesionUsuario.getId(), sesionUsuario);
  }
  @OnMessage
  public void recibeMensaje(String mensaje, Session sesionUsuario)
    throws IOException {
    String nombre = (String) sesionUsuario.getUserProperties().get("nombre");
    if (nombre == null) {
      sesionUsuario.getUserProperties().put("nombre", mensaje);
      sesionUsuario.getBasicRemote()
                .sendText(mensajeJson("Sistema","Te has conectado como " + mensaje));
    } else {
      Collection<Session> sesiones = usuarios.values();
      Iterator it = sesiones.iterator();
      while(it.hasNext()) {
        Session estaSesion = (Session) it.next();
        estaSesion.getBasicRemote()
                .sendText(mensajeJson(nombre,mensaje));
      }
    }
  }
  @OnClose
  public void cerrarSocket(Session sesionUsuario) {
    usuarios.remove(sesionUsuario.getId());
  }
  public String mensajeJson(String nombre, String mensaje) {
    StringWriter sTringWriter = new StringWriter();
    JsonWriter jsonWriter = Json.createWriter(sTringWriter);
    JsonObject jSON = Json.createObjectBuilder()
                .add("mensaje", nombre + ": " + mensaje + "").build();
    jsonWriter.write(jSON);
    return sTringWriter.toString();
  }
}
```

El método "mensajeJson(String nombre, String mensaje)" propuesto envía al cliente:

```
"mensaje" = {
        nombre : mensaje
}
```

En el lado del cliente tendremos un fichero "index.html" (HTML estándar) con el siguiente contenido:

```
<html>
  <head>
```

```
    <title>Chatsocket</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script type="text/javascript" src="js/jquery-3.1.1.min.js"></script>
    <script type="text/javascript">
      var websocket = new WebSocket("ws://localhost:7001/Chat-war/chatsocket");
      websocket.onmessage = function procesarRespuesta(mensaje) {
        var datosJson = $.parseJSON(mensaje.data);
        if (datosJson.mensaje!==null) {
          mensajetextoarea.value =
                mensajetextoarea.value += datosJson.mensaje + "\n";
        }
      };
      function enviar() {
        websocket.send(mensajetexto.value);
        mensajetexto.value = "";
      }
    </script>
  </head>
  <body>
    <textarea id="mensajetextoarea"
        readonly="readonly" rows="10" cols="45"></textarea> <br />
        <input type="text" id="mensajetexto" size="50" />
        <input type="button" value="Enviar" onClick="enviar();" />
  </body>
</html>
```

Una buena práctica para el lector es implementar este ejemplo en un proyecto web (WAR) llamado *Chat-war.**

*Este ejercicio es de dominio público y se puede encontrar en diversos blogs y ejemplos en videotutoriales. En nuestro caso, el lector puede seguir un videotutorial con este ejemplo en "https://www.youtube.com/watch?v=BikL52HYaZg".

8.4 Ejercicio 7

Se propone al lector la finalización de la aplicación web del blog, implementando mediante *websockets* las operaciones de:

- Escribir un *post*.
- Añadir un comentario a un *post*.
- Establecer una conversación de chat con un usuario.
- Enviar mensajes de chat.

De modo que mediante JSF realizaremos la gestión de plantillas y la entrada o salida del blog *(login/logout)*.

Los mensajes que se enviarán serán los siguientes:

- Del servidor al cliente.
  - Mensaje de *flash* con información o error.
  - Mensaje con el estado del cliente.
- Del cliente al servidor.
  - Mensaje descriptivo de la operación del usuario.

En el servidor, tendremos las siguientes clases:

- Para enviar mensaje *flash*:

```
package estadocliente;
import java.io.Serializable;
public class MensajeFlash implements Serializable {
   private String operacion;
   private String titulo;
   private String error;
   private String mensaje;
   private String nombreUsuario;
   public MensajeFlash() {
      this.operacion="mensajeflash";
   }
   public String getOperacion() {
      return operacion;
   }
   public void setOperacion(String operacion) {
      this.operacion = operacion;
   }
   public String getTitulo() {
      return titulo;
   }
   public void setTitulo(String titulo) {
      this.titulo = titulo;
   }
   public String getError() {
      return error;
   }
   public void setError(String error) {
      this.error = error;
   }
   public String getMensaje() {
      return mensaje;
   }
```

```
  public void setMensaje(String mensaje) {
    this.mensaje = mensaje;
  }
  public String getNombreUsuario() {
    return nombreUsuario;
  }
  public void setNombreUsuario(String nombreUsuario) {
    this.nombreUsuario = nombreUsuario;
  }
}
```

Los campos que recibirá el cliente serán tratados con GSon. O sea, cuando se envíe un mensaje *flash* al cliente le llegará así:

```
var mensaje = {
        "operacion" : "mensajeflash",
        "titulo" : "El mejor blog del mundo",
        "error" : "Mensaje de error obtenido desde el back-end",
        "mensaje" : "Información de cualquier tipo no errónea",
        "nombreUsuario" : "nombre del usuario logado"
};
```

La clase que plasmará el estado de un usuario será la siguiente:

```
package estadocliente;
import java.util.List;
public class EstadoCliente {
  private String operacion;
  private String nombreUsuario;
  private List<Usuario> usuarios;
  private List<Post> posts;
  private String chatActivo; // nombreUsuario.
  private List<MensajeChat> chats; // Listado de chats con el usuario activo.
  public EstadoCliente(){
    this.operacion = "actualiza";
  }
  public String getOperacion() {
    return operacion;
  }
  public void setOperacion(String operacion) {
    this.operacion = operacion;
  }
  public String getNombreUsuario() {
    return nombreUsuario;
  }
  public void setNombreUsuario(String nombreUsuario) {
    this.nombreUsuario = nombreUsuario;
  }
```

```
  public List<Usuario> getUsuarios() {
    return usuarios;
  }
  public void setUsuarios(List<Usuario> usuarios) {
    this.usuarios = usuarios;
  }
  public List<Post> getPosts() {
    return posts;
  }
  public void setPosts(List<Post> posts) {
    this.posts = posts;
  }
  public String getChatActivo() {
    return chatActivo;
  }
  public void setChatActivo(String chatActivo) {
    this.chatActivo = chatActivo;
  }
  public List<MensajeChat> getChats() {
    return chats;
  }
  public void setChats(List<MensajeChat> chats) {
    this.chats = chats;
  }
}
```

Todas son autodescriptivas menos la de usuario. La de usuario contendrá el nombre, si es el propio usuario de la sesión (booleano), y si se está chateando con él (booleano). La clase quedaría tal cual sigue:

```
package estadocliente;
import java.io.Serializable;
public class Usuario implements Serializable {
  private String nombre;
  private boolean yoMismo;
  private boolean chateando;
  public Usuario(){}
  public String getNombre() {
    return nombre;
  }
  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
  public boolean isYoMismo() {
    return yoMismo;
  }
```

```
  public void setYoMismo(boolean yoMismo) {
    this.yoMismo = yoMismo;
  }
  public boolean isChateando() {
    return chateando;
  }
  public void setChateando(boolean chateando) {
    this.chateando = chateando;
  }
}
```

Cada vez que ocurra un evento registrado por el servidor, se enviará a cada cliente la información de su estado. Al mismo tiempo, cada cliente enviará al servidor cada cuatro segundos una señal indicando que está vivo.

Del cliente al servidor enviaremos el siguiente objeto:

```
var operacion = {
        definicion: "",
        nombreUsuario: "",
        idPost: "",
        contenidoTexto: "",
        usuarioChat: ""
    };
```

Donde *definición* puede tomar alguno de los siguientes valores:

- keepalive.
- login.
- escribePost.
- escribeComentario.
- chateacon.
- enviachat.

Y los demás campos toman valores según la definición de la operación. Se propone al lector terminar de implementar el blog con toda la funcionalidad especificada:

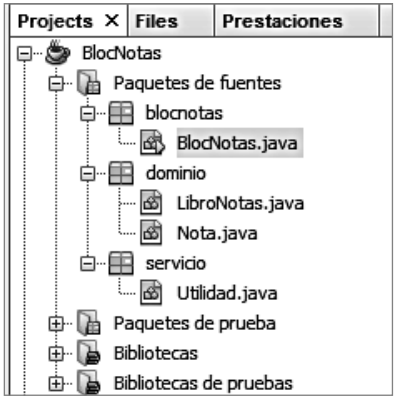
- *Login* (JSF).
- Escribir un *post (websockets)*.
- Añadir un comentario a un *post (websockets)*.
- Establecer una conversación de chat con un usuario *(websockets)*.
- Enviar mensajes de chat *(websockets)*.
- *Logout* (JSF).

Ejercicios resueltos

CAPÍTULO 9

9.1 Ejercicio 1

La estructura de carpetas del proyecto es la siguiente:



- Modelo del dominio. Clase "Nota.java":

```
package dominio;
import java.util.Calendar;
public class Nota {
   public Nota(){}
   private String texto;
   private Calendar fecha;
   public String getTexto() {
      return texto;
   }
```

```
    public void setTexto(String texto) {
        this.texto = texto;
    }
    public Calendar getFecha() {
        return fecha;
    }
    public void setFecha(Calendar fecha) {
        this.fecha = fecha;
    }
}
```

- Clase "LibroNotas.java":

```
package dominio;
import java.util.ArrayList;
import java.util.List;
import servicio.Utilidad;
public class LibroNotas {
    private List<Nota> notas;
    public LibroNotas() {
        this.notas = new ArrayList<>();
    }
    public List<Nota> getNotas() {
        return notas;
    }
    public void setNotas(List<Nota> notas) {
        this.notas = notas;
    }
    public boolean anadirNota(Nota nota) {
        this.notas.add(nota);
        this.notas = Utilidad.ordenarNotasBurbujaFecha(this.notas);
        return true;
    }
}
```

- Paquete de servicio. Clase "Utilidad.java":

```
package servicio;
import dominio.Nota;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.List;
public class Utilidad {
    private static BufferedReader br = null;
```

```
public static void mostrarMenu() {
   System.out.println("1.- Insertar nota".);
   System.out.println("2.- Ver notas".);
   System.out.println("3.- Salir".);
   System.out.println("****");
}
public static Nota tomarNota() {
   mostrarTexto("--NUEVA NOTA--");
   Calendar fecha = recogerFecha("-FECHA-");
   mostrarTexto("-TEXTO-");
   String texto = recogerTexto("Nota");
   Nota n = new Nota();
   n.setFecha(fecha);
   n.setTexto(texto);
   return n;
}
public static void mostrarNotas(List<Nota> notas) {
   if ( (notas!=null) && (!notas.isEmpty()) ) {
      System.out.println("--LISTADO DE NOTAS--");
      int i = 1;
      for(Nota n : notas) {
         Calendar fecha = n.getFecha();
         String texto = n.getTexto();
         String fechaCadena = convertirFechaCadena(fecha);
         System.out.println("Nota " + i + ". Fecha: " + fechaCadena
               + ". Nota: " + texto + "".);
         i++;
      }
   } else {
      mostrarTexto(": No existen notas que mostrar".);
   }
}
public static void mostrarTexto(String textoMostrar) {
   System.out.println(textoMostrar);
}
public static String recogerTexto(String textoMostrar) {
   if (br == null) br = new BufferedReader(new InputStreamReader(System.in));
   String entrada = null;
   while (entrada == null) {
      System.out.print(textoMostrar + ": ");
      try {
         entrada = br.readLine();
      } catch(Exception e) {}
   }
   return entrada;
}
```

```
public static int recogerValor(int min, int max, String textoMostrar) {
   if (br == null) br = new BufferedReader(new InputStreamReader(System.in));
   String entrada;
   int value = -1000;
   while ((value < min) || (value > max)) {
      System.out.print(textoMostrar + ": ");
      try {
         entrada = br.readLine();
         value = Integer.parseInt(entrada);
      } catch (Exception e) {}
   }
   return value;
}
public static Calendar recogerFecha(String textoMostrar) {
   if (br == null) br = new BufferedReader(new InputStreamReader(System.in));
   System.out.println(textoMostrar);
   mostrarMeses();
   int mes = recogerValor(1, 12, "Seleccione mes");
   int ano = recogerValor(2016,2030,"Seleccione año entre 2016 y 2030");
   int dia = 0;
   if ((mes == 1) || (mes == 3) || (mes == 5) || (mes == 7) || (mes == 8)
         || (mes == 10) || (mes == 12)) {
      dia = recogerValor(1, 31, "Seleccione el día entre 1 y 31");
   } else {
      if (mes == 2) {
         if(esBisiesto(ano)) {
            dia = recogerValor(1, 29, "Seleccione el día entre 1 y 29");
         } else {
            dia = recogerValor(1, 28, "Seleccione el día entre 1 y 28");
         }
      } else {
         dia = recogerValor(1, 30, "Seleccione el día entre 1 y 30");
      }
   }
   mes = mes - 1;
   return new GregorianCalendar(ano,mes,dia);
}
private static String convertirFechaCadena(Calendar fecha) {
   int ano = fecha.get(Calendar.YEAR);
   int mes = fecha.get(Calendar.MONTH) + 1;
   int dia = fecha.get(Calendar.DAY_OF_MONTH);
   String anoString = "" + ano;
   String mesString = (mes < 10) ? "0" + mes : "" + mes;
   String diaString = (dia < 10) ? "0" + dia : "" + dia;
   String fechaString = diaString + "/" + mesString + "/" + anoString;
   return fechaString;
}
```

```
    private static void mostrarMeses() {
        System.out.println("Los meses del año"); // Días
        System.out.println("1.- Enero".); // 31
        System.out.println("2.- Febrero".); // 28 o 29
        System.out.println("3.- Marzo".); // 31
        System.out.println("4.- Abril".); // 30
        System.out.println("5.- Mayo".); // 31
        System.out.println("6.- Junio".); // 30
        System.out.println("7.- Julio".); // 31
        System.out.println("8.- Agosto".); // 31
        System.out.println("9.- Septiembre".); // 30
        System.out.println("10.- Octubre".); // 31
        System.out.println("11.- Noviembre".); // 30
        System.out.println("12.- Diciembre".); // 31
    }
    private static boolean esBisiesto(int ano) {
        boolean bisiesto = false;
        int mod4 = ano%4;
        int mod100 = ano%100;
        int mod400 = ano%400;
        if ( (mod4 == 0) && ( (!(mod100==0)) || (mod400 == 0))  ) {
            bisiesto = true;
        }
        return bisiesto;
    }
    public static List<Nota> ordenarNotasBurbujaFecha(List<Nota> listaNotas) {
        if ((listaNotas != null) && (listaNotas.size() >= 2)) {
            int nNotas = listaNotas.size();
            for(int i = 1 ; i < nNotas ; i++) {
                int max = nNotas - i;
                for(int j = 0; j < max; j++) {
                    Nota notaJ = listaNotas.get(j);
                    Nota notaJ1 = listaNotas.get(j+1);
                    Calendar fechaJ = notaJ.getFecha();
                    Calendar fechaJ1 = notaJ1.getFecha();
                    int comp = fechaJ.compareTo(fechaJ1);
                    if (comp > 0) {
                        listaNotas.set(j, notaJ1);
                        listaNotas.set(j+1, notaJ);
                    }
                }
            }
        }
        return listaNotas;
    }
}
```

- Paquete "blocnotas". Clase "BlocNotas.java". Método "main()".

```
package blocnotas;
import dominio.LibroNotas;
import dominio.Nota;
import servicio.Utilidad;
public class BlocNotas {
   public static void main(String[] args) {
      int valor;
      LibroNotas libro = new LibroNotas();
      Utilidad.mostrarTexto("BLOC DE NOTAS");
      do {
         Utilidad.mostrarMenu();
         valor = Utilidad.recogerValor(1, 3, "Seleccione una opción");
         if (valor == 1) {
            Nota n = Utilidad.tomarNota();
            libro.anadirNota(n);
         } else if (valor == 2) {
            Utilidad.mostrarNotas(libro.getNotas());
         }
      } while (valor != 3);
   }
}
```

9.2 Ejercicio 2

Nuestro paquete de datos "data" en el proyecto del blog queda de la siguiente forma:

El código de las entidades "Usuario", "ComentaPost", "Chat" y "ChatActual" se muestran a continuación.

- "Usuario.java":

```
package data;
import java.io.Serializable;
```

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class Usuario implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String nombreUsuario;
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public String getNombreUsuario() {
      return nombreUsuario;
   }
   public void setNombreUsuario(String nombreUsuario) {
      this.nombreUsuario = nombreUsuario;
   }
   @Override
   public int hashCode() {
      int hash = 0;
      hash += (id != null ? id.hashCode() : 0);
      return hash;
   }
   @Override
   public boolean equals(Object object) {
      if (!(object instanceof Usuario)) {
         return false;
      }
      Usuario other = (Usuario) object;
      if ((this.id == null && other.id != null)
            || (this.id != null && !this.id.equals(other.id))) {
         return false;
      }
      return true;
   }
   @Override
   public String toString() {
      return "data.Usuario[ id=" + id + " ]";
   }
}
```

- "ComentaPost.java":

```
package data;
import java.io.Serializable;
import java.util.Calendar;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
@Entity
public class ComentaPost implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private Calendar fechaInstante;
   private String comentario;
   @ManyToOne
   private Usuario usuario;
   @ManyToOne
   private Post post;
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public Calendar getFechaInstante() {
      return fechaInstante;
   }
   public void setFechaInstante(Calendar fechaInstante) {
      this.fechaInstante = fechaInstante;
   }
   public String getComentario() {
      return comentario;
   }
   public void setComentario(String comentario) {
      this.comentario = comentario;
   }
   public Usuario getUsuario() {
      return usuario;
   }
   public void setUsuario(Usuario usuario) {
      this.usuario = usuario;
   }
```

```
    public Post getPost() {
        return post;
    }
    public void setPost(Post post) {
        this.post = post;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof ComentaPost)) {
            return false;
        }
        ComentaPost other = (ComentaPost) object;
        if ((this.id == null && other.id != null) ||
                (this.id != null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "data.ComentaPost[ id=" + id + " ]";
    }
}
```

- "Chat.java":

```
package data;
import java.io.Serializable;
import java.util.Calendar;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
@Entity
public class Chat implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
```

```
    private Calendar fechaInstante;
    private String contenido;
    @ManyToOne
    private Usuario enviadoPor;
    @ManyToOne
    private Usuario enviadoA;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public Calendar getFechaInstante() {
        return fechaInstante;
    }
    public void setFechaInstante(Calendar fechaInstante) {
        this.fechaInstante = fechaInstante;
    }
    public String getContenido() {
        return contenido;
    }
    public void setContenido(String contenido) {
        this.contenido = contenido;
    }
    public Usuario getEnviadoPor() {
        return enviadoPor;
    }
    public void setEnviadoPor(Usuario enviadoPor) {
        this.enviadoPor = enviadoPor;
    }
    public Usuario getEnviadoA() {
        return enviadoA;
    }
    public void setEnviadoA(Usuario enviadoA) {
        this.enviadoA = enviadoA;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Chat)) {
            return false;
        }

        Chat other = (Chat) object;
        if ((this.id == null && other.id != null)
```

```
            || (this.id != null && !this.id.equals(other.id))) {
        return false;
      }
      return true;
    }
    @Override
    public String toString() {
      return "data.Chat[ id=" + id + " ]";
    }
}
```

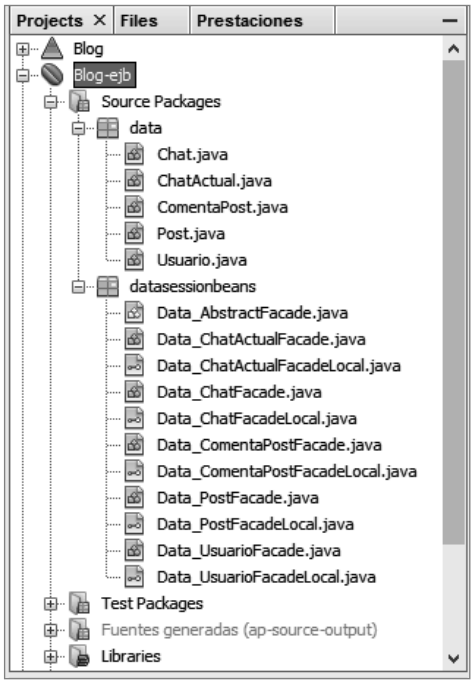
- "ChatActual.java":

```
package data;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.OneToOne;
@Entity
public class ChatActual implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @OneToOne
    private Usuario usuario;
    @ManyToOne
    private Usuario chateaCon;
    public Long getId() {
      return id;
    }
    public void setId(Long id) {
      this.id = id;
    }
    public Usuario getUsuario() {
      return usuario;
    }
    public void setUsuario(Usuario usuario) {
      this.usuario = usuario;
    }

    public Usuario getChateaCon() {
      return chateaCon;
    }
    public void setChateaCon(Usuario chateaCon) {
      this.chateaCon = chateaCon;
    }
    @Override
    public int hashCode() {
```

```
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof ChatActual)) {
            return false;
        }
        ChatActual other = (ChatActual) object;
        if ((this.id == null && other.id != null)
                || (this.id != null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "data.ChatActual[ id=" + id + " ]";
    }
}
```

9.3 Ejercicio 3

La estructura del proyecto con los paquetes "data" y "datasessionbeans".



- Interfaz "Data_ComentaPostFacadeLocal.java":

```
package datasessionbeans;
import data.ComentaPost;
import java.util.List;
import javax.ejb.Local;
@Local
public interface Data_ComentaPostFacadeLocal {
   void create(ComentaPost comentaPost);
   void edit(ComentaPost comentaPost);
   void remove(ComentaPost comentaPost);
   ComentaPost find(Object id);
   List<ComentaPost> findAll();
   List<ComentaPost> findRange(int[] range);
   int count();
}
```

- *Stateless* "Data_ComentaPostFacade.java":

```
package datasessionbeans;
import data.ComentaPost;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Stateless
public class Data_ComentaPostFacade extends Data_AbstractFacade<ComentaPost>
      implements Data_ComentaPostFacadeLocal {
   @PersistenceContext(unitName = "Blog-ejbPU")
   private EntityManager em;
   @Override
   protected EntityManager getEntityManager() {
      return em;
   }
   public Data_ComentaPostFacade() {
      super(ComentaPost.class);
   }
}
```

- Interfaz "Data_ChatFacadeLocal.java":

```
package datasessionbeans;
import data.Chat;
import java.util.List;
import javax.ejb.Local;
@Local
public interface Data_ChatFacadeLocal {
   void create(Chat chat);
```

```
   void edit(Chat chat);
   void remove(Chat chat);
   Chat find(Object id);
   List<Chat> findAll();
   List<Chat> findRange(int[] range);
   int count();
}
```

- *Stateless* "Data_ChatFacade.java":

```
package datasessionbeans;
import data.Chat;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Stateless
public class Data_ChatFacade extends Data_AbstractFacade<Chat>
      implements Data_ChatFacadeLocal {
   @PersistenceContext(unitName = "Blog-ejbPU")
   private EntityManager em;
   @Override
   protected EntityManager getEntityManager() {
      return em;
   }
   public Data_ChatFacade() {
      super(Chat.class);
   }
}
```

- Interfaz "Data_ChatActualFacadeLocal.java":

```
package datasessionbeans;
import data.ChatActual;
import java.util.List;
import javax.ejb.Local;
@Local
public interface Data_ChatActualFacadeLocal {
   void create(ChatActual chatActual);
   void edit(ChatActual chatActual);
   void remove(ChatActual chatActual);
   ChatActual find(Object id);
   List<ChatActual> findAll();
   List<ChatActual> findRange(int[] range);
   int count();
}
```

- *Stateless* "Data_ChatActualFacade.java":

```
package datasessionbeans;
import data.ChatActual;
import javax.ejb.Stateless;
```

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Stateless
public class Data_ChatActualFacade extends Data_AbstractFacade<ChatActual>
      implements Data_ChatActualFacadeLocal {
   @PersistenceContext(unitName = "Blog-ejbPU")
   private EntityManager em;
   @Override
   protected EntityManager getEntityManager() {
      return em;
   }
   public Data_ChatActualFacade() {
      super(ChatActual.class);
   }
}
```

9.4 Ejercicio 4

En los apartados y ejercicios previos hemos implementado las clases de los paquetes "data" y "datasessionbeans". Aun así, tendremos que modificar algunas fachadas de acceso a datos para adaptarlas a las necesidades del servicio. Veamos el contenido de los paquetes "domainmodel", "exceptions", "util", de las fachadas de acceso a datos modificadas, y del paquete "service".

Paquete "domainmodel":

- Clase "BlogPost.java":

```
package domainmodel;
import java.io.Serializable;
import java.util.List;
public class BlogPost implements Serializable {
   private Long id;
   private String fechaInstante;
   private String contenido;
   private String autor;
   private List<ComentarioPost> comentarios;
   public BlogPost() {}
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public String getFechaInstante() {
      return fechaInstante;
   }
   public void setFechaInstante(String fechaInstante) {
      this.fechaInstante = fechaInstante;
   }
   public String getContenido() {
      return contenido;
   }
   public void setContenido(String contenido) {
      this.contenido = contenido;
   }
   public String getAutor() {
      return autor;
   }
   public void setAutor(String autor) {
      this.autor = autor;
   }
   public List<ComentarioPost> getComentarios() {
      return comentarios;
   }
   public void setComentarios(List<ComentarioPost> comentarios) {
      this.comentarios = comentarios;
   }
}
```

- Clase "ComentarioPost.java":

```
package domainmodel;
import java.io.Serializable;
public class ComentarioPost implements Serializable {
   private String fechaInstante;
   private String comentario;
   private String usuario;
   public ComentarioPost() {}
   public String getFechaInstante() {
      return fechaInstante;
   }
   public void setFechaInstante(String fechaInstante) {
      this.fechaInstante = fechaInstante;
   }
   public String getComentario() {
      return comentario;
   }
   public void setComentario(String comentario) {
      this.comentario = comentario;
   }
   public String getUsuario() {
      return usuario;
   }
   public void setUsuario(String usuario) {
      this.usuario = usuario;
   }
}
```

- Clase "ChateaCon.java":

```
package domainmodel;
import java.io.Serializable;
public class ChateaCon implements Serializable {
   private String usuario;
   private String con;
   public ChateaCon() {}
   public String getUsuario() {
      return usuario;
   }
   public void setUsuario(String usuario) {
      this.usuario = usuario;
   }
   public String getCon() {
      return con;
   }
   public void setCon(String con) {
      this.con = con;
   }
}
```

- Clase "MensajeChat.java":

```
package domainmodel;
import java.io.Serializable;
public class MensajeChat implements Serializable {
   private String fechaInstante;
   private String contenido;
   private String enviadoPor;
   private String enviadoA;
   public MensajeChat() {}
   public String getFechaInstante() {
      return fechaInstante;
   }
   public void setFechaInstante(String fechaInstante) {
      this.fechaInstante = fechaInstante;
   }
   public String getContenido() {
      return contenido;
   }
   public void setContenido(String contenido) {
      this.contenido = contenido;
   }
   public String getEnviadoPor() {
      return enviadoPor;
   }
   public void setEnviadoPor(String enviadoPor) {
      this.enviadoPor = enviadoPor;
   }
   public String getEnviadoA() {
      return enviadoA;
   }
   public void setEnviadoA(String enviadoA) {
      this.enviadoA = enviadoA;
   }
}
```

Paquete "exceptions":

- Excepción "BlogException.java":

```
package exceptions;
public class BlogException extends Exception {
   public BlogException() {}
   public BlogException(String msg) {
      super(msg);
   }
}
```

Paquete "util". En él implementamos una clase con métodos de utilidad:

- Clase "ServiceUtility.java":

```
package util;
import data.Chat;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.List;
public class ServiceUtility {
   public static String getFechaInstanteCadena(Calendar fecha) {
      int ano = fecha.get(Calendar.YEAR);
      int mes = fecha.get(Calendar.MONTH) + 1;
      int dia = fecha.get(Calendar.DAY_OF_MONTH);
      int hora = fecha.get(Calendar.HOUR_OF_DAY);
      int minutos = fecha.get(Calendar.MINUTE);
      int segundos = fecha.get(Calendar.SECOND);
      String anoString = "" + ano;
      String mesString = (mes < 10)  ?  "0" + mes : "" + mes;
      String diaString = (dia < 10) ? "0" + dia : "" + dia;
      String horaString = (hora < 10) ? "0" + hora : "" + hora;
      String minutosString = (minutos < 10) ? "0" + minutos : "" + minutos;
      String segundosString = (segundos < 10) ? "0" + segundos : "" + segundos;
      String fechaString = diaString + "/" + mesString + "/" + anoString
            + " " + horaString + ":" + minutosString + ":" + segundosString;
      return fechaString;
   }
   public static Calendar getFechaInstanteCalendar(String fechaInstante) {
      String[] partes = fechaInstante.split(" ");
      String fecha = partes[0];
      String instante = partes[1];
      String[] partesFecha = fecha.split("/");
      String[] partesInstante = instante.split(":");
      String diaString = partesFecha[0];
      String mesString = partesFecha[1];
      String anoString = partesFecha[2];
      String horaString = partesInstante[0];
      String minutosString = partesInstante[1];
      String segundosString = partesInstante[2];
      int dia = Integer.parseInt(diaString);
      int mes = Integer.parseInt(mesString);
      int ano = Integer.parseInt(anoString);
      int hora = Integer.parseInt(horaString);
      int minutos = Integer.parseInt(minutosString);
      int segundos = Integer.parseInt(segundosString);
      Calendar fInstante = new GregorianCalendar(ano, mes, dia,
            hora, minutos, segundos);
      return fInstante;
   }
```

```
   public static List<Chat> ordenarChatsBurbuja(List<Chat> chats) {
      int nChats = chats.size();
      for(int i=1;i<nChats;i++) {
         int max = nChats-i;
         for(int j=0;j<max;j++) {
            Chat chatJ = chats.get(j);
            Chat chatJ1 = chats.get(j+1);
            Calendar fechaInstanteJ = chatJ.getFechaInstante();
            Calendar fechaInstanteJ1 = chatJ1.getFechaInstante();
            int comp = fechaInstanteJ.compareTo(fechaInstanteJ1);
            if (comp<0) {
               chats.set(j, chatJ1);
               chats.set(j+1, chatJ);
            }
         }
      }
      return chats;
   }
}
```

Veamos ahora las fachadas de acceso a datos que hemos modificado para poder tener acceso a los datos que necesitamos desde la capa de servicio.

- Interfaz "Data_UsuarioFacadeLocal.java":

```
package datasessionbeans;
import data.Usuario;
import java.util.List;
import javax.ejb.Local;
@Local
public interface Data_UsuarioFacadeLocal {
   void create(Usuario usuario);
   void edit(Usuario usuario);
   void remove(Usuario usuario);
   Usuario find(Object id);
   List<Usuario> findAll();
   List<Usuario> findRange(int[] range);
   int count();
   // Añadido.
   public Long encontrarUsuarioporNombre(String nombreUsuario);
}
```

Hemos añadido un método para encontrar el "id" de un usuario por su nombre. Si no existe se devuelve -1. A continuación, la implementación:

- Clase "Data_UsuarioFacade.java":

```
package datasessionbeans;
import data.Usuario;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
@Stateless
public class Data_UsuarioFacade extends Data_AbstractFacade<Usuario>
      implements Data_UsuarioFacadeLocal {
   @PersistenceContext(unitName = "Blog-ejbPU")
   private EntityManager em;
   @Override
   protected EntityManager getEntityManager() {
      return em;
   }
   public Data_UsuarioFacade() {
      super(Usuario.class);
   }
   @Override
   public Long encontrarUsuarioporNombre(String nombreUsuario) {
      TypedQuery<Usuario> query = this.getEntityManager().createQuery(
            "SELECT u FROM Usuario u WHERE u.nombreUsuario = :nombreUsuario",
            data.Usuario.class);
      query.setParameter("nombreUsuario", nombreUsuario);
      List<Usuario> usuarios = query.getResultList();
      if (usuarios.isEmpty() || usuarios.size() > 1) {
         return null;
      } else {
         return usuarios.get(0).getId();
      }
   }
}
```

- Interfaz "Data_ComentaPostFacadeLocal.java":

```
package datasessionbeans;
import data.ComentaPost;
import java.util.List;
import javax.ejb.Local;
@Local
public interface Data_ComentaPostFacadeLocal {
```

```
    void create(ComentaPost comentaPost);
    void edit(ComentaPost comentaPost);
    void remove(ComentaPost comentaPost);
    ComentaPost find(Object id);
    List<ComentaPost> findAll();
    List<ComentaPost> findRange(int[] range);
    int count();
    // Añadido.
    public List<ComentaPost> obtenerLosComentariosdePost(Long postId);
}
```

Hemos añadido un método para obtener los comentarios de un *post*:

- Clase "Data_ComentaPostFacade.java":

```
package datasessionbeans;
import data.ComentaPost;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
@Stateless
public class Data_ComentaPostFacade extends Data_AbstractFacade<ComentaPost>
      implements Data_ComentaPostFacadeLocal {
    @PersistenceContext(unitName = "Blog-ejbPU")
    private EntityManager em;
    @Override
    protected EntityManager getEntityManager() {
      return em;
    }
    public Data_ComentaPostFacade() {
      super(ComentaPost.class);
    }
    @Override
    public List<ComentaPost> obtenerLosComentariosdePost(Long postId) {
      TypedQuery<ComentaPost> query = this.getEntityManager().createQuery(
            "SELECT cPost FROM ComentaPost cPost WHERE cPost.post.id = "
                  + postId +" ORDER BY cPost.fechaInstante DESC",
            data.ComentaPost.class);
      List<ComentaPost> comentaPosts = query.getResultList();
      return comentaPosts;
    }
}
```

- Interfaz "Data_ChatActualFacadeLocal.java":

```
package datasessionbeans;
import data.ChatActual;
import java.util.List;
import javax.ejb.Local;
@Local
public interface Data_ChatActualFacadeLocal {
   void create(ChatActual chatActual);
   void edit(ChatActual chatActual);
   void remove(ChatActual chatActual);
   ChatActual find(Object id);
   List<ChatActual> findAll();
   List<ChatActual> findRange(int[] range);
   int count();
   // Añadido.
   public Long existeChatUsuario(Long idUsuario);
}
```

Hemos añadido un método que devuelve el "id" de un "ChatActual" del usuario cuyo "id" se pasa por parámetro. La implementación es la siguiente:

- Clase "Data_ChatActualFacade.java":

```
package datasessionbeans;
import data.ChatActual;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
@Stateless
public class Data_ChatActualFacade extends Data_AbstractFacade<ChatActual>
      implements Data_ChatActualFacadeLocal {
   @PersistenceContext(unitName = "Blog-ejbPU")
   private EntityManager em;
   @Override
   protected EntityManager getEntityManager() {
      return em;
   }
   public Data_ChatActualFacade() {
      super(ChatActual.class);
   }
   @Override
   public Long existeChatUsuario(Long idUsuario) {
      TypedQuery<ChatActual> query = this.getEntityManager().createQuery(
            "SELECT chact FROM ChatActual chact WHERE chact.usuario.id = " + idUsuario,
```

```
            data.ChatActual.class);
      List<ChatActual> chats = query.getResultList();
      if (chats.isEmpty() || chats.size() > 1) {
         return null;
      } else {
         return chats.get(0).getId();
      }
   }
}
```

- Interfaz "Data_ChatFacadeLocal.java":

```
package datasessionbeans;
import data.Chat;
import java.util.List;
import javax.ejb.Local;
@Local
public interface Data_ChatFacadeLocal {
   void create(Chat chat);
   void edit(Chat chat);
   void remove(Chat chat);
   Chat find(Object id);
   List<Chat> findAll();
   List<Chat> findRange(int[] range);
   int count();
   // Añadido.
   public List<Chat> chatsDeUsuarioAOtro(Long idUsuario1, Long idUsuario2);
}
```

Añadimos un método para obtener un listado de chats (mensajes) de un usuario a otro (de forma unidireccional, enviados desde usuario 1 a usuario 2). La implementación a continuación:

- Clase "Data_ChatFacade.java":

```
package datasessionbeans;
import data.Chat;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
@Stateless
```

```
public class Data_ChatFacade extends Data_AbstractFacade<Chat>
      implements Data_ChatFacadeLocal {
   @PersistenceContext(unitName = "Blog-ejbPU")
   private EntityManager em;
   @Override
   protected EntityManager getEntityManager() {
      return em;
   }
   public Data_ChatFacade() {
      super(Chat.class);
   }
   @Override
   public List<Chat> chatsDeUsuarioAOtro(Long idUsuario1, Long idUsuario2) {
      TypedQuery<Chat> query = this.getEntityManager()
            .createQuery("SELECT cht FROM Chat cht WHERE "
                  + "cht.enviadoPor.id = " + idUsuario1 + " AND "
                  + "cht.enviadoA.id = " + idUsuario2 +
                  " ORDER BY cht.fechaInstante DESC", data.Chat.class);
      List<Chat> chats = query.getResultList();
      return chats;
   }
}
```

Finalmente, acudimos a la capa de servicio, paquete "service". En dicha capa acabará el *back-end* de nuestra aplicación (si no tenemos en cuenta los servicios web).

- Interfaz "IBlog.java":

```
package service;
import domainmodel.*;
import exceptions.BlogException;
import java.util.List;
public interface IBlog {
   public boolean crearUsuario(String usuario) throws BlogException;
   public Long existeUsuario(String usuario) throws BlogException;
   public List<String> obtenerTodosUsuarios() throws BlogException;
   public BlogPost crearPost(BlogPost post) throws BlogException;
   public BlogPost obtenerPostPorId(Long postId) throws BlogException;
   public ComentarioPost anadirComentarioPost(BlogPost post,
         ComentarioPost comentario) throws BlogException;
   public boolean abreChatCon(ChateaCon chatea) throws BlogException;
   public String obtenerChatActualUsuario(String usuario) throws BlogException;
   public MensajeChat enviaMensajeChat(MensajeChat chat) throws BlogException;
   public List<BlogPost> obtenerTodosLosPosts() throws BlogException;
   public List<MensajeChat> obtenerTodosLosMensajesChat(String usuario1,
         String usuario2) throws BlogException;
}
```

- Interfaz "BlogServiceLocal.java":

```
package service;
import javax.ejb.Local;
@Local
public interface BlogServiceLocal extends IBlog {}
```

- Interfaz "BlogServiceRemote.java":

```
package service;
import javax.ejb.Remote;
@Remote
public interface BlogServiceRemote extends IBlog  {}
```

Finalmente, y para acabar el ejercicio, vemos la clase que implementa la capa de servicio con un *stateless session bean*, implementando las interfaces local y remota.

- Clase EJB *stateless* "BlogService.java":

```
package service;
import data.*;
import datasessionbeans.*;
import domainmodel.*;
import exceptions.BlogException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.List;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import util.ServiceUtility;
@Stateless
public class BlogService implements BlogServiceLocal, BlogServiceRemote {
   @EJB
   private Data_UsuarioFacadeLocal data_UsuarioFacade;
   @EJB
   private Data_PostFacadeLocal data_PostFacade;
   @EJB
   private Data_ComentaPostFacadeLocal data_ComentaPostFacade;
   @EJB
   private Data_ChatFacadeLocal data_ChatFacade;
   @EJB
   private Data_ChatActualFacadeLocal data_ChatActualFacade;
   @Override
   public boolean crearUsuario(String usuario) throws BlogException {
      if (usuario == null || usuario.equals("")) {
         throw new BlogException("BlogService - crear usuario: "
               + "No se ha pasado usuario a insertar en la BD");
```

```
    }
    Usuario u = new Usuario();
    u.setNombreUsuario(usuario);
    try {
      this.data_UsuarioFacade.create(u);
    } catch(Exception e) {
      throw new BlogException("BlogService - crear usuario: "
          + "El nombre del usuario ya existe en la BD");
    }
    return true;
  }
  @Override
  public Long existeUsuario(String usuario) throws BlogException {
    if (usuario == null || usuario.equals("")) {
      throw new BlogException("BlogService - existe usuario: "
          + "No se ha pasado usuario a recuperar de la BD");
    }
    Long usuarioId = this.data_UsuarioFacade.encontrarUsuarioporNombre(usuario);
    if (usuarioId == null) {
      return -1L;
    } else {
      return usuarioId;
    }
  }
  @Override
  public List<String> obtenerTodosUsuarios() throws BlogException {
    List<Usuario> usuarios = this.data_UsuarioFacade.findAll();
    if (usuarios == null) throw new BlogException("BlogService - obtener usuarios: "
        + "No se pudieron obtener todos los usuarios de la BD");
    List<String> listado = new ArrayList<>();
    for(Usuario u: usuarios) {
      String esteUsuario = u.getNombreUsuario();
      listado.add(esteUsuario);
    }
    return listado;
  }
  @Override
  public BlogPost crearPost(BlogPost post) throws BlogException {
    if (post==null) {
      throw new BlogException("BlogService - crear post: "
          + "No se ha pasado post a introducir de la BD");
    }
    String autor = post.getAutor();
    String contenido = post.getContenido();
    if (autor==null || autor.equals("")) throw
        new BlogException("BlogService - crear post: "
```

```
            + "No se ha pasado autor del post a introducir de la BD");
    if (contenido == null || contenido.equals("")) throw
          new BlogException("BlogService - crear post: "
            + "No se ha pasado contenido del post a introducir de la BD");
    Long autorId = this.existeUsuario(autor);
    if (autorId == -1L) throw new BlogException("BlogService - crear post: "
            + "El autor del post no existe en la base de datos");
    Usuario u  = data_UsuarioFacade.find(autorId);
    Post p = new Post();
    p.setAutor(u);
    p.setContenido(contenido);
    p.setFechaInstante(new GregorianCalendar());
    try {
       this.data_PostFacade.create(p);
    } catch(Exception e) {
       throw new BlogException("BlogService - crear post: "
            + "No se pudo crear el post en la BD");
    }
    Calendar fechaInstante = p.getFechaInstante();
    String fInstante = ServiceUtility.getFechaInstanteCadena(fechaInstante);
    post.setFechaInstante(fInstante);
    post.setId(p.getId());
    post.setComentarios(new ArrayList<ComentarioPost>());
    return post;
  }
  @Override
  public BlogPost obtenerPostPorId(Long postId) throws BlogException {
    if (postId == null) {
       throw new BlogException("BlogService - obtener post por id: "
            + "No se ha pasado correctamente el id");
    }
    Post p = null;
    try {
       p = this.data_PostFacade.find(postId);
    } catch(Exception e) {
       throw new BlogException("BlogService - obtener post por id: "
            + "No se pudo obtener el post de la BD");
    }
    BlogPost bPost = new BlogPost();
    bPost.setAutor(p.getAutor().getNombreUsuario());
    bPost.setContenido(p.getContenido());
    bPost.setId(p.getId());
    bPost.setFechaInstante(ServiceUtility
```

```
            .getFechaInstanteCadena(p.getFechaInstante()));
    List<ComentaPost> comentaPosts = null;
    try {
      comentaPosts = this.data_ComentaPostFacade
            .obtenerLosComentariosdePost(postId);
    } catch(Exception e) {
      throw new BlogException("BlogService - obtener post por id: "
            + "No se pudo obtener el post de la BD");
    }
    List<ComentarioPost> comentarios = new ArrayList<>();
    for (ComentaPost cPost: comentaPosts) {
      ComentarioPost comentario = new ComentarioPost();
      comentario.setComentario(cPost.getComentario());
      comentario.setFechaInstante(ServiceUtility
            .getFechaInstanteCadena(cPost.getFechaInstante()));
      comentario.setUsuario(cPost.getUsuario().getNombreUsuario());
      comentarios.add(comentario);
    }
    bPost.setComentarios(comentarios);
    return bPost;
  }
  @Override
  public ComentarioPost anadirComentarioPost(BlogPost post,
        ComentarioPost comentario) throws BlogException {
    if (post == null)
      throw new BlogException("BlogService - añadir comentario post: "
            + "No se ha pasado post existente en la BD");
    if (comentario == null)
      throw new BlogException("BlogService - añadir comentario post: "
            + "No se ha pasado comentario a añadir a la BD");
    String contenido = comentario.getComentario();
    if (contenido==null || contenido.equals("")) {
      throw new BlogException("BlogService - añadir comentario post: "
            + "No se ha pasado comentario a añadir a la BD");
    }
    Long postId = post.getId();
    Post p = data_PostFacade.find(postId);
    if (p == null) {
      throw new BlogException("BlogService - añadir comentario post: "
            + "No se ha pasado post existente en la BD");
    }
    String autorComentario = comentario.getUsuario();
    Long usuarioId = existeUsuario(autorComentario);
    if (usuarioId == -1L) {
      throw new BlogException("BlogService - añadir comentario post: "
            + "No se ha pasado autor del comentario existente en la BD");
```

```
        }
        Usuario u = data_UsuarioFacade.find(usuarioId);
        if (u == null) {
            throw new BlogException("BlogService - añadir comentario post: "
                    + "No se ha pasado autor del comentario existente en la BD");
        }
        ComentaPost cp = new ComentaPost();
        cp.setPost(p);
        cp.setUsuario(u);
        cp.setFechaInstante(new GregorianCalendar());
        cp.setComentario(contenido);
        try {
            this.data_ComentaPostFacade.create(cp);
        } catch(Exception e) {
            throw new BlogException("BlogService - añadir comentario post: "
                    + "No se pudo añadir el comentario al post");
        }
        Calendar fechaInstante = cp.getFechaInstante();
        String fInstante = ServiceUtility.getFechaInstanteCadena(fechaInstante);
        comentario.setFechaInstante(fInstante);
        return comentario;
    }
    @Override
    public boolean abreChatCon(ChateaCon chatea) throws BlogException {
        if (chatea == null) {
            throw new BlogException("BlogService - actualiza chat destino: "
                    + "el objeto no se pasó bien");
        }
        String usuario1 = chatea.getUsuario();
        String usuario2 = chatea.getCon();
        if (usuario1 == null || usuario1.equals(""))
            throw new BlogException("BlogService - actualiza chat destino: "
                    + "el usuario activo no se pasó correctamente");
        if (usuario2 == null || usuario2.equals(""))
            throw new BlogException("BlogService - actualiza chat destino: "
                    + "el usuario pasivo no se pasó correctamente");
        Long usuario1Id = existeUsuario(usuario1);
        Long usuario2Id = existeUsuario(usuario2);
        if ((usuario1Id == -1L) || (usuario2Id == -1L)) {
            throw new BlogException("BlogService - actualiza chat destino: "
                    + "alguno de los usuarios no existe en la BD");
        }
        Usuario u1 = null;
        Usuario u2 = null;
        try {
            u1 = this.data_UsuarioFacade.find(usuario1Id);
```

```
            u2 = this.data_UsuarioFacade.find(usuario2Id);
        } catch(Exception e) {
            throw new BlogException("BlogService - actualiza chat destino: "
                    + "alguno de los usuarios no existe en la BD");
        }
        if (u1 == null || u2 == null) {
            throw new BlogException("BlogService - actualiza chat destino: "
                    + "alguno de los usuarios no existe en la BD");
        }
        ChatActual chatAct = new ChatActual();
        chatAct.setUsuario(u1);
        chatAct.setChateaCon(u2);
        Long chatActualId = this.data_ChatActualFacade.existeChatUsuario(usuario1Id);
        try {
            if (chatActualId == null) {
                this.data_ChatActualFacade.create(chatAct);
            } else {
                chatAct.setId(chatActualId);
                this.data_ChatActualFacade.edit(chatAct);
            }
        } catch(Exception e) {
            throw new BlogException("BlogService - actualiza chat destino: "
                    + "no se pudo llevar a cabo la operación");
        }
        return true;
    }
    @Override
    public String obtenerChatActualUsuario(String usuario) throws BlogException {
        if ( (usuario == null) || (usuario.equals("")) ) {
            throw new BlogException("BlogService - obtener chat actual usuario: "
                    + "no se pasó bien el nombre del usuario");
        }
        Long usuarioId = existeUsuario(usuario);
        if (usuarioId == -1L) {
            throw new BlogException("BlogService - obtener chat actual usuario: "
                    + "no se pasó bien el nombre del usuario");
        }
        Long chatActualId = this.data_ChatActualFacade.existeChatUsuario(usuarioId);
        if (chatActualId == null) {
            return "";
        }
        String nombreUsuario = null;
        ChatActual chatAct = null;
        try {
            chatAct = this.data_ChatActualFacade.find(chatActualId);
            nombreUsuario = chatAct.getUsuario().getNombreUsuario();
```

```
    } catch(Exception e) {
      throw new BlogException("BlogService - obtener chat actual usuario: "
          + "no se pudo obtener");
    }
    if (!nombreUsuario.equals(usuario)) {
      throw new BlogException("BlogService - obtener chat actual usuario: "
          + "no se pudo obtener");
    }
    return chatAct.getChateaCon().getNombreUsuario();
  }
  @Override
  public MensajeChat enviaMensajeChat(MensajeChat chat) throws BlogException {
    if (chat == null) {
      throw new BlogException("BlogService - envía mensaje chat: "
          + "el mensaje no se seteó correctamente");
    }
    String contenidoChat = chat.getContenido();
    String usuario1 = chat.getEnviadoPor();
    String usuario2 = chat.getEnviadoA();
    if ((contenidoChat == null) || (contenidoChat.equals(""))
        || (usuario1 == null) || (usuario1.equals(""))
        || (usuario2 == null) || (usuario2.equals("")) ) {
      throw new BlogException("BlogService - envía mensaje chat: "
          + "el mensaje no se seteó correctamente");
    }
    Long usuario1Id = existeUsuario(usuario1);
    Long usuario2Id = existeUsuario(usuario2);
    if ((usuario1Id == -1L) || (usuario2Id == -1L)) {
      throw new BlogException("BlogService - envía mensaje chat: "
          + "alguno de los usuarios no existe en la BD");
    }
    Usuario u1 = null;
    Usuario u2 = null;
    try {
      u1 = this.data_UsuarioFacade.find(usuario1Id);
      u2 = this.data_UsuarioFacade.find(usuario2Id);
    } catch(Exception e) {
      throw new BlogException("BlogService - envía mensaje chat: "
          + "alguno de los usuarios no existe en la BD");
    }
    if (u1 == null || u2 == null) {
      throw new BlogException("BlogService - envía mensaje chat: "
          + "alguno de los usuarios no existe en la BD");
    }
    Chat mensaje = new Chat();
    mensaje.setContenido(contenidoChat);
```

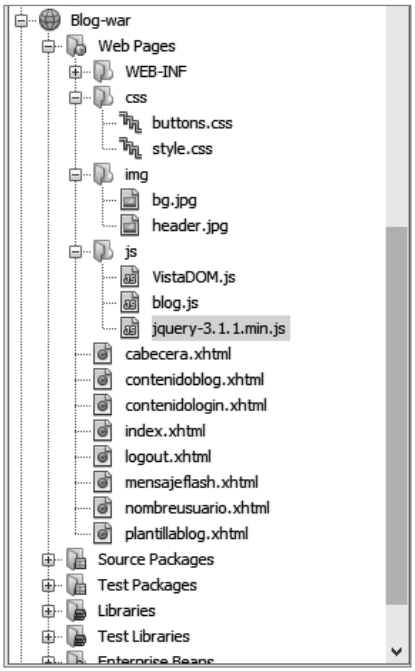
```
    mensaje.setEnviadoPor(u1);
    mensaje.setEnviadoA(u2);
    mensaje.setFechaInstante(new GregorianCalendar());
    try {
      this.data_ChatFacade.create(mensaje);
    } catch(Exception e) {
      throw new BlogException("BlogService - envía mensaje chat: "
          + "no se pudo registrar el mensaje en la BD");
    }
    Calendar fechaInstante = mensaje.getFechaInstante();
    String fInstante = ServiceUtility.getFechaInstanteCadena(fechaInstante);
    chat.setFechaInstante(fInstante);
    return chat;
  }
  @Override
  public List<BlogPost> obtenerTodosLosPosts() throws BlogException {
    List<Post> posts = this.data_PostFacade.findAll();
    int nPosts = posts.size();
    List<BlogPost> blogPosts = new ArrayList<>();
    // Ordenamos al revés para poner el último al comienzo.
    for(int j=nPosts-1;j>=0;j--) {
      Post estePost = posts.get(j);
      BlogPost esteBlogPost = new BlogPost();
      esteBlogPost.setAutor(estePost.getAutor().getNombreUsuario());
      esteBlogPost.setContenido(estePost.getContenido());
      esteBlogPost.setFechaInstante(ServiceUtility
          .getFechaInstanteCadena(estePost.getFechaInstante()));
      Long postId = estePost.getId();
      esteBlogPost.setId(postId);
      List<ComentaPost> comentaPosts = null;
      try {
        comentaPosts =
            this.data_ComentaPostFacade.obtenerLosComentariosdePost(postId);
      } catch(Exception e) {
        throw new BlogException("BlogService - obtener los comentarios de post: "
          + "no se pudieron obtener");
      }
      List<ComentarioPost> comentarios = new ArrayList<>();
      for(ComentaPost cPost : comentaPosts) {
        ComentarioPost cp = new ComentarioPost();
        cp.setUsuario(cPost.getUsuario().getNombreUsuario());
        cp.setFechaInstante(ServiceUtility
            .getFechaInstanteCadena(cPost.getFechaInstante()));
        cp.setComentario(cPost.getComentario());
        comentarios.add(cp);
```

```
        }
        esteBlogPost.setComentarios(comentarios);
        blogPosts.add(esteBlogPost);
      }
      return blogPosts;
    }
    @Override
    public List<MensajeChat> obtenerTodosLosMensajesChat(String usuario1,
        String usuario2) throws BlogException {
      if ((usuario1 == null) || (usuario1.equals(""))
        || (usuario2 == null) || (usuario2.equals("")) ) {
        throw new BlogException("BlogService - obtener todos los mensajes de chat: "
            + "los usuarios no se setearon correctamente");
      }
      Long usuario1Id = existeUsuario(usuario1);
      Long usuario2Id = existeUsuario(usuario2);
      if ((usuario1Id == -1L) || (usuario2Id == -1L)) {
        throw new BlogException("BlogService - obtener todos los mensajes de chat: "
            + "los usuarios no se setearon correctamente");
      }
      Usuario u1 = null;
      Usuario u2 = null;
      try {
        u1 = this.data_UsuarioFacade.find(usuario1Id);
        u2 = this.data_UsuarioFacade.find(usuario2Id);
      } catch(Exception e) {
        throw new BlogException("BlogService - obtener todos los mensajes de chat: "
            + "alguno de los usuarios no existe en la BD");
      }
      if (u1 == null || u2 == null) {
        throw new BlogException("BlogService - obtener todos los mensajes de chat: "
            + "alguno de los usuarios no existe en la BD");
      }
      List<Chat> todosChats = null;
      try {
        List<Chat> chats1 =
            this.data_ChatFacade.chatsDeUsuarioAOtro(usuario1Id, usuario2Id);
        List<Chat> chats2 =
            this.data_ChatFacade.chatsDeUsuarioAOtro(usuario2Id, usuario1Id);
        todosChats = chats1;
        todosChats.addAll(chats2);
        todosChats = ServiceUtility.ordenarChatsBurbuja(todosChats);
      } catch(Exception e) {
        throw new BlogException("BlogService - obtener todos los mensajes de chat: "
            + "no se pudieron obtener los mensajes de chat");
```

```
        }
        List<MensajeChat> mensajes = new ArrayList<MensajeChat>();
        for(Chat esteChat : todosChats) {
            MensajeChat mensaje = new MensajeChat();
            mensaje.setContenido(esteChat.getContenido());
            mensaje.setFechaInstante(ServiceUtility
                    .getFechaInstanteCadena(esteChat.getFechaInstante()));
            mensaje.setEnviadoPor(esteChat.getEnviadoPor().getNombreUsuario());
            mensaje.setEnviadoA(esteChat.getEnviadoA().getNombreUsuario());
            mensajes.add(mensaje);
        }
        return mensajes;
    }
}
```

9.5 Ejercicio 5

Comenzamos a realizar la implementación del módulo web (WAR). La estructura de carpetas es la siguiente:



En este apartado veremos la implementación propuesta de la hoja de estilos CSS propia (la de botones no la hemos desarrollado nosotros). Veremos el *bean* administrado situado en el paquete "managedbeans" (del módulo WAR) y todas las páginas JSF.

Comencemos viendo la hoja de estilos propia:

- Hoja de estilos "style.css":

```
@CHARSET "UTF-8";
/* Hoja de estilos para el ejercicio del blog */
* {
   margin: 0;
   padding: 0;
}
html {
   height: 100%;
}
html body {
   background: #fff url(/Blog-war/img/bg.jpg) repeat-y top center;
   font-family: Arial, Helvetica, sans-serif;
   font-size: 12px;
   line-height: 17px;
   color: #222;
   height: 100%;
}
html body div#nombreusuario {
   display : none;
}
html body div#mensajeflash {
   position: absolute;
   left: 50%;
   top: 120px;
   transform: translate(-50%, -50%);
   -webkit-transform: translate(-50%, -50%);
   z-index: 10;
}
html body div#mensajeflash div {
   padding: 10px;
   font-weight: bold;
   -webkit-border-radius: 10px;
   -moz-border-radius: 10px;
   border-radius: 10px;
}
html body div#mensajeflash div.oculto {
   display: none;
}
html body div#mensajeflash div.visible {
   display: block;
}
html body div#mensajeflash div.mensaje {
```

```
    background-color: green;
    color: white;
    border: 2px solid white;
}
html body div#mensajeflash div.error {
    background-color: red;
    color: yellow;
    border: 2px solid yellow;
}
html body div#contenedor {
    margin: 0 auto;
    width: 800px;
    height: 100%;
}
html body div#contenedor div#cabecera {
    height: 90px;
    margin: 0;
    background: #fff url(/Blog-war/img/header.jpg) no-repeat;
}
html body div#contenedor div#cabecera h1 {
    font-size: 25px;
    letter-spacing: -1px;
    padding: 25px 0 0 20px;
    color: #fff;
}
html body div#contenedor div#cabecera h2 {
    font-size: 18px;
    color: #336699;
    padding: 3px 0 0 20px;
    letter-spacing: -1px;
    font-weight: 100;
}
html body div#contenedor div#cabecera div#muestranombreusuario {
    display:block;
    margin-left: 20px;
}
html body div#contenedor div#cabecera div#logout {
    float: right;
    display: inline-block;
    margin-right: 40px;
}
html body div#contenedor div#cabecera div.clear {
    clear: both;
}
html body div#contenedor div#contenido {
    height: 100%;
    padding: 20px;
```

```
}
html body div#contenedor div#contenido div#login,
html body div#contenedor div#contenido div#registro {
   height: 100%;
}
html body div#contenedor div#contenido div#login table,
html body div#contenedor div#contenido div#registro table {
   border: 2px solid gray;
   -webkit-border-radius: 10px;
   -moz-border-radius: 10px;
   border-radius: 10px;
   margin: 150px auto;
}
html body div#contenedor div#contenido div#login table tr td,
html body div#contenedor div#contenido div#registro table tr td {
   padding: 10px;
}
html body div#contenedor div#contenido div.red {
   float: left;
   height: 100%;
   border: 1px solid gray;
}
html body div#contenedor div#contenido div#izquierdo {
   width: 150px;
}
html body div#contenedor div#contenido div#izquierdo div#titulousuarios {
   padding-left: 10px;
   line-height: 26px;
   background: #B9B9B9;
   color: #fff;
   font-size: 13px;
   font-weight: bold;
}
html body div#contenedor div#contenido div#izquierdo div#listadousuarios {
   height: 95%;
   overflow: scroll;
}
html body div#contenedor div#contenido div#izquierdo div#listadousuarios div.usuario {
   border: 1px solid black;
   color: gray;
   text-align: center;
   padding: 3px;
}
html body div#contenedor div#contenido div#central {
   width: 400px;
}
```

```
html body div#contenedor div#contenido div#central div.tituloposts {
   text-align: center;
   line-height: 26px;
   background: #B9B9B9;
   color: #fff;
   font-size: 16px;
   font-weight: bold;
}
html body div#contenedor div#contenido div#central div#listadoposts {
   height: 95%;
   overflow: scroll;
}
html body div#contenedor div#contenido div#central div#listadoposts div.escribepost,
html body div#contenedor div#contenido div#central div#listadoposts div.post {
   border: 1px solid black;
   color: gray;
   text-align: justify;
   padding: 3px 3px 3px 5px;
}
html body div#contenedor div#contenido div#central div#listadoposts div.post div.escribeco-
mentariopost {
   padding: 0 10px 0 10px;
   border: 1px solid gray;
}
html body div#contenedor div#contenido div#central div#listadoposts div.post div.comentar-
iopost {
   padding-left: 20px;
   border: 1px solid gray;
}
html body div#contenedor div#contenido div#derecho {
   width: 200px;
}
html body div#contenedor div#contenido div#derecho div.titulo {
   text-align: center;
   background: #B9B9B9;
   color: #fff;
   font-weight: bold;
}
html body div#contenedor div#contenido div#derecho div#contenedorchats {
   height: 50%;
   overflow: scroll;
}
html body div#contenedor div#contenido div#derecho div#contenedorchats div.mensajechat {
   color: gray;
   padding: 3px;
   text-align: left;
```

```
}
html body div#contenedor div#contenido div#derecho div#enviochat {
   height: 30%;
}
html body div#contenedor div#contenido div#derecho div#enviochat span#usuariochat {
   display: none;
}
html body div#contenedor div#contenido div#derecho div#enviochat textarea {
   width: 99%;
   height: 50%;
}
html body div#contenedor div#contenido div div.clear {
   clear: both;
}
textarea {
   resize: none;
   width: 100%;
}
```

- *Bean* gestionado en el paquete "managedbeans". Los métodos que gestionarán la entrada y la salida del usuario del sistema son "accesoUsuario()" y "salidaUsuario()". Cuando realizamos el *login* y la consulta y se crea el usuario en la base de datos, es necesaria la inserción del EJB BlogService.

```
package managedbeans;
import exceptions.BlogException;
import javax.inject.Named;
import java.io.Serializable;
import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import service.BlogServiceLocal;
@ManagedBean
@Named(value = "usuarioBean")
@SessionScoped
public class UsuarioBean implements Serializable {
   @EJB
   private BlogServiceLocal blogService;
   private String titulo;
   private String nombreUsuario;
   private String mensaje;
   private String error;
   private boolean enviaNombreUsuario;
   private boolean enviaDatosFlash;
   private boolean enviaCabecera;
   private boolean inicio;
```

```
private boolean blog;
public UsuarioBean() {
   this.titulo = "El mejor blog del mundo";
   this.nombreUsuario = "";
   this.mensaje = "";
   this.error = "";
   this.enviaDatosFlash = true;
   this.enviaCabecera = true;
   this.inicio = true;
   this.enviaNombreUsuario = true;
   this.blog = false;
}
public void accesoUsuario() {
   try {
      String nombreSinEspacios = this.nombreUsuario.trim();
      if (!nombreSinEspacios.equals("")) {
         Long usuarioId = blogService.existeUsuario(this.nombreUsuario);
         boolean exito = false;
         if(usuarioId.equals(-1L)) {
            // Hay que crear el usuario.
            exito = blogService.crearUsuario(this.nombreUsuario);
         } else {
            exito = true;
         }
         if(exito) {
            // Hemos accedido. Procedemos a ajustar los valores del bean.
            this.mensaje = "Bienvenido";
            this.error = "";
            this.inicio = false;
            this.blog = true;
         }
      } else {
         this.nombreUsuario = "";
         this.mensaje = "";
         throw new BlogException("Error al introducir el nombre");
      }
   } catch (BlogException ex) {
      this.error = ex.getMessage();
   }
}
public void salidaUsuario() {
   this.nombreUsuario = "";
   this.mensaje = "";
   this.error = "";
   this.enviaDatosFlash = true;
   this.enviaCabecera = true;
```

```
        this.inicio = true;
        this.enviaNombreUsuario = true;
        this.blog = false;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getNombreUsuario() {
        return nombreUsuario;
    }
    public void setNombreUsuario(String nombreUsuario) {
        this.nombreUsuario = nombreUsuario;
    }
    public String getMensaje() {
        return mensaje;
    }
    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }
    public String getError() {
        return error;
    }
    public void setError(String error) {
        this.error = error;
    }
    public boolean isEnviaDatosFlash() {
        return enviaDatosFlash;
    }
    public void setEnviaDatosFlash(boolean enviaDatosFlash) {
        this.enviaDatosFlash = enviaDatosFlash;
    }
    public boolean isEnviaCabecera() {
        return enviaCabecera;
    }
    public void setEnviaCabecera(boolean enviaCabecera) {
        this.enviaCabecera = enviaCabecera;
    }
    public boolean isInicio() {
        return inicio;
    }
    public void setInicio(boolean inicio) {
        this.inicio = inicio;
```

```
    }
    public boolean isEnviaNombreUsuario() {
        return enviaNombreUsuario;
    }
    public void setEnviaNombreUsuario(boolean enviaNombreUsuario) {
        this.enviaNombreUsuario = enviaNombreUsuario;
    }
    public boolean isBlog() {
        return blog;
    }
    public void setBlog(boolean blog) {
        this.blog = blog;
    }
}
```

Y las páginas JSF planteadas al lector son las siguientes:

- "plantillablog.xhtml":

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <ui:composition>
    <h:head>
      <title>Blog</title>
      <meta charset="UTF-8" />
      <script type="text/javascript" src="js/jquery-3.1.1.min.js"></script>
      <script type="text/javascript" src="js/VistaDOM.js"></script>
      <script type="text/javascript" src="js/blog.js"></script>
      <link rel="stylesheet" type="text/css" href="css/style.css" />
      <link rel="stylesheet" type="text/css" href="css/buttons.css" />
    </h:head>
    <h:body>
      <ui:fragment rendered="#{usuarioBean.enviaNombreUsuario}">
        <ui:insert name="nombreusuario">
          <ui:include src="/nombreusuario.xhtml" />
        </ui:insert>
      </ui:fragment>
      <ui:fragment rendered="#{usuarioBean.enviaDatosFlash}">
        <ui:insert name="mensajeflash">
          <ui:include src="/mensajeflash.xhtml" />
        </ui:insert>
      </ui:fragment>
      <div id="contenedor">
```

```
            <ui:fragment rendered="#{usuarioBean.enviaCabecera}">
              <ui:insert name="cabecera">
                <ui:include src="/cabecera.xhtml" />
              </ui:insert>
            </ui:fragment>
            <ui:fragment rendered="#{usuarioBean.inicio}">
              <ui:insert name="contenidologin">
                <ui:include src="/contenidologin.xhtml" />
              </ui:insert>
            </ui:fragment>
            <ui:fragment rendered="#{usuarioBean.blog}">
              <ui:insert name="contenidoblog">
                <ui:include src="/contenidoblog.xhtml" />
              </ui:insert>
            </ui:fragment>
          </div>
        </h:body>
      </ui:composition>
</html>
```

- "nombreusuario.xhtml" (elemento que permanecerá oculto mediante CSS):

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <body>
    <ui:composition>
      <div id="nombreusuario">
        <h:outputText value="#{usuarioBean.nombreUsuario}" />
      </div>
    </ui:composition>
  </body>
</html>
```

- "mensajeflash.xhtml" (elemento que ocultaremos o mostraremos con JavaScript):

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
```

```
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <body>
    <ui:composition>
      <div id="mensajeflash">
        <div class="error">
          <h:outputText value="#{usuarioBean.error}" />
        </div>
        <div class="mensaje">
          <h:outputText value="#{usuarioBean.mensaje}" />
        </div>
      </div>
    </ui:composition>
  </body>
</html>
```

- "cabecera.xhtml":

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <body>
    <ui:composition>
      <div id="cabecera">
        <ui:fragment rendered="#{usuarioBean.blog}">
          <ui:insert name="logout">
            <ui:include src="/logout.xhtml" />
          </ui:insert>
        </ui:fragment>
        <h1>
          <h:outputText value="Blog JEE" />
        </h1>
        <h2>
          <h:outputText value="#{usuarioBean.titulo}" />
        </h2>
        <div class="clear"></div>
      </div>
    </ui:composition>
  </body>
</html>
```

- "logout.xhtml":

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://xmlns.jcp.org/jsf/html"
     xmlns:f="http://java.sun.com/jsf/core"
     xmlns:ui="http://java.sun.com/jsf/facelets">
   <body>
      <ui:composition>
         <div id="logout">
            <h:form>
               <h:commandLink class="button gray" value="Logout"
                        action="#{usuarioBean.salidaUsuario()}" />
            </h:form>
         </div>
      </ui:composition>
   </body>
</html>
```

- "contenidologin.xhtml":

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://xmlns.jcp.org/jsf/html"
     xmlns:f="http://java.sun.com/jsf/core"
     xmlns:ui="http://java.sun.com/jsf/facelets">
   <body>
      <ui:composition>
         <div id="contenido">
            <div id="login">
               <h:form>
                  <h:panelGrid id="login" columns="2">
                     <h:outputLabel value="Nombre de usuario: " />
                     <h:inputText id="username" label="username"
                        value="#{usuarioBean.nombreUsuario}" />
                     <h:commandButton class="button gray" value="Enviar"
                        type="submit" action="#{usuarioBean.accesoUsuario()}"  />
                     <h:commandButton class="button gray"
                        value="Restablecer" type="reset" />
                  </h:panelGrid>
               </h:form>
            </div>
```

```
        </div>
      </ui:composition>
  </body>
</html>
```

- "contenidoblog.xhtml":

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <body>
    <ui:composition>
      <div id="contenido">
        <div id="izquierdo" class="red">
          <div id="titulousuarios">
            <h:outputText value="Usuarios" />
          </div>
          <div id="listadousuarios"></div>
        </div>
        <div id="central" class="red">
          <div class="tituloposts">
            <h:outputText value="Posts" />
          </div>
          <div id="listadoposts">
            <div class="tituloposts">
              <h:outputText value="Escribe un Post" />
            </div>
            <div class="escribepost">
              <h:outputText value="¿En qué estás pensando?" />
              <br />
              <h:form>
                <h:inputTextarea id="escribepost" label="escribepost">
                </h:inputTextarea>
              </h:form>
              <br />
              <h:link class="escribepost button gray" value="Enviar" />
            </div>
            <div class="tituloposts">
              <h:outputText value="Posts tuyos y de los demás usuarios" />
            </div>
          </div>
        </div>
```

```
            <div id="derecho" class="red">
              <div class="titulo">
                <h:outputText value="Chat" />
              </div>
              <div id="contenedorchats"></div>
              <div id="enviochat">
                <div class="titulo">
                  <h:outputText value="Área de Chat" />
                </div>
                <h:form>
                  <h:inputTextarea id="campochat" ></h:inputTextarea>
                </h:form>
                <br />
                <span id="usuariochat" nombreusuario="" />
                <h:link id="enviarchat" class="button gray" value="Enviar" />
              </div>
            </div>
          </div>
        </ui:composition>
      </body>
    </html>
```

9.6 Ejercicio 6

Debemos implementar las clases "PostResource.java" y "ComentarioResource.java".

- Clase "PostResource.java":

```
package webservices;
import domainmodel.BlogPost;
import exceptions.BlogException;
import java.util.Hashtable;
import java.util.List;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response.Status;
```

```
import service.BlogServiceRemote;
@Path("/post")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class PostResource {
   private BlogServiceRemote blogService = null;
   @Context
   private UriInfo context;
   public PostResource() {
      if (blogService == null) {
         try {
            blogService = obtenerReferenciaBlogService();
         } catch (NamingException ex) {
            System.out.println("Error crítico. Sin acceso al back-end");
         }
      }
   }
   @GET
   public List<BlogPost> getPosts() {
      try {
         return blogService.obtenerTodosLosPosts();
      } catch (BlogException ex) {
         throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
      }
   }
   @POST
   public BlogPost addPost(BlogPost post) {
      if (post == null) {
         throw new WebApplicationException(Status.BAD_REQUEST);
      }
      String autor = post.getAutor();
      String contenido = post.getContenido();
      if (autor == null || autor.equals("") || contenido == null ||
            contenido.equals("")) {
         throw new WebApplicationException(Status.BAD_REQUEST);
      }
      Long autorId = null;
      try {
         autorId = blogService.existeUsuario(autor);
      } catch(Exception e) {
         throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
      }
      if (autorId == -1L) {
         throw new WebApplicationException(Status.BAD_REQUEST);
      }
      try {
```

```
            post = blogService.crearPost(post);
        } catch(Exception e) {
            throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
        }
        return post;
    }
    @GET
    @Path("/{postId}")
    public BlogPost getPostById(@PathParam("postId") String postId) {
        if (postId==null || postId.equals("")) {
            throw new WebApplicationException(Status.BAD_REQUEST);
        }
        Long postIdL = null;
        try {
            postIdL = Long.valueOf(postId);
        } catch(Exception e) {
            throw new WebApplicationException(Status.BAD_REQUEST);
        }
        BlogPost post = null;
        try {
            post = blogService.obtenerPostPorId(postIdL);
            if (post == null) throw new WebApplicationException(Status.NOT_FOUND);
        } catch(Exception e) {
            throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
        }
        return post;
    }
    @Path("/{postId}/comentario")
    public ComentarioResource getComentarioResource() {
        return new ComentarioResource();
    }
    private BlogServiceRemote obtenerReferenciaBlogService()
            throws NamingException {
        Hashtable hash = new Hashtable();
        hash.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                "weblogic.jndi.WLInitialContextFactory");
        hash.put(javax.naming.Context.PROVIDER_URL, "t3://localhost:7001");
        javax.naming.Context ctx = new InitialContext(hash);
        BlogServiceRemote bsr = (BlogServiceRemote)
                ctx.lookup("java:global.Blog.Blog-ejb.BlogService!"
                + "service.BlogServiceRemote");
        return bsr;
    }
```

```
}
```

Y para el subrecurso tenemos:

- Clase "ComentarioResource.java":

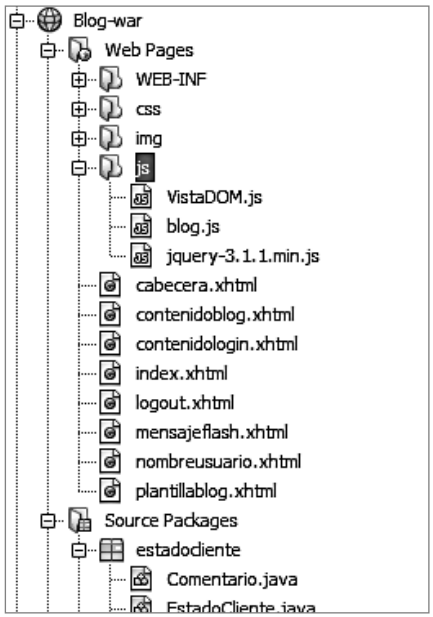
```
package webservices;
import domainmodel.BlogPost;
import domainmodel.ComentarioPost;
import java.util.Hashtable;
import java.util.List;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import service.BlogServiceRemote;
@Path("/")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class ComentarioResource {
   private BlogServiceRemote blogService = null;
   @Context
   private UriInfo context;
   public ComentarioResource() {
      if (blogService == null) {
         try {
            blogService = obtenerReferenciaBlogService();
         } catch (NamingException ex) {
            System.out.println("Error crítico. Sin acceso al back-end");
         }
      }
   }
   @GET
   public List<ComentarioPost> getComentarios(@PathParam("postId") String postId) {
      if (postId==null || postId.equals("")) {
         throw new WebApplicationException(Response.Status.BAD_REQUEST);
      }
      Long postIdL = null;
      try {
```

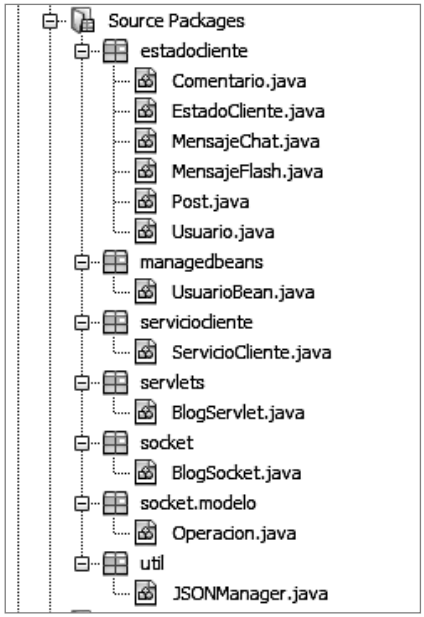
```
        postIdL = Long.valueOf(postId);
      } catch(Exception e) {
        throw new WebApplicationException(Response.Status.BAD_REQUEST);
      }
      BlogPost post = null;
      try {
        post = blogService.obtenerPostPorId(postIdL);
        if (post == null)
          throw new WebApplicationException(Response.Status.NOT_FOUND);
      } catch(Exception e) {
        throw new WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
      }
      return post.getComentarios();
    }
    @POST
    public ComentarioPost anadeComentarioPost(@PathParam("postId")
        String postId, ComentarioPost comentario) {
      if (postId==null || postId.equals("")) {
        throw new WebApplicationException(Response.Status.BAD_REQUEST);
      }
      Long postIdL = null;
      try {
        postIdL = Long.valueOf(postId);
      } catch(Exception e) {
        throw new WebApplicationException(Response.Status.BAD_REQUEST);
      }
      BlogPost post = null;
      try {
        post = blogService.obtenerPostPorId(postIdL);
        if (post == null)
          throw new WebApplicationException(Response.Status.NOT_FOUND);
      } catch(Exception e) {
        throw new WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
      }
      try {
        comentario = blogService.anadirComentarioPost(post, comentario);
      } catch(Exception e) {
        throw new WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
      }
      return comentario;
    }
    private BlogServiceRemote obtenerReferenciaBlogService()
        throws NamingException {
      Hashtable hash = new Hashtable();
      hash.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
          "weblogic.jndi.WLInitialContextFactory");
```

```
        hash.put(javax.naming.Context.PROVIDER_URL, "t3://localhost:7001");
        javax.naming.Context ctx = new InitialContext(hash);
        BlogServiceRemote bsr = (BlogServiceRemote)
              ctx.lookup("java:global.Blog.Blog-ejb.BlogService!"
              + "service.BlogServiceRemote");
        return bsr;
    }
}
```

9.7 Ejercicio 7

Se nos propone dar toda la funcionalidad al ejercicio del blog. Tenemos un punto de acceso al *back-end* en el EJB *stateless* BlogService que implementa las interfaces local y remota. Veamos cómo queda nuestro módulo WAR, en el cual hemos implementado ya las páginas JSF y el *bean* gestionado de sesión "UsuarioBean". La estructura de carpetas del módulo WAR es la siguiente:





En el planteamiento del ejercicio hemos visto cómo quedan las clases:

- MensajeFlash.java.
- EstadoCliente.java.
- Usuario.java.
- JSONManager.java.

Todas del paquete "estadocliente". Dicho paquete es el que contiene la definición en el lado del servidor de las vistas de cada uno de los clientes (cada cliente tendrá una vista particular). El resto de clases que componen el paquete "estadocliente" son las siguientes:

- "Post.java":

```
package estadocliente;
import java.io.Serializable;
import java.util.List;
public class Post implements Serializable {
   private String id;
   private String fechaInstante;
   private String autor;
   private String contenido;
   private List<Comentario> comentarios;
   public Post(){}
   public String getId() {
      return id;
   }
   public void setId(String id) {
      this.id = id;
   }
   public String getFechaInstante() {
      return fechaInstante;
   }
   public void setFechaInstante(String fechaInstante) {
      this.fechaInstante = fechaInstante;
   }
   public String getAutor() {
      return autor;
   }
   public void setAutor(String autor) {
      this.autor = autor;
   }
   public String getContenido() {
      return contenido;
   }
   public void setContenido(String contenido) {
      this.contenido = contenido;
   }
   public List<Comentario> getComentarios() {
      return comentarios;
   }
   public void setComentarios(List<Comentario> comentarios) {
      this.comentarios = comentarios;
   }
```

```
}
```

- "Comentario.java":

```
package estadocliente;
import java.io.Serializable;
public class Comentario implements Serializable {
   private String nombreUsuario;
   private String fechaInstante;
   private String contenido;
   public Comentario() {}
   public String getNombreUsuario() {
      return nombreUsuario;
   }
   public void setNombreUsuario(String nombreUsuario) {
      this.nombreUsuario = nombreUsuario;
   }
   public String getFechaInstante() {
      return fechaInstante;
   }
   public void setFechaInstante(String fechaInstante) {
      this.fechaInstante = fechaInstante;
   }
   public String getContenido() {
      return contenido;
   }
   public void setContenido(String contenido) {
      this.contenido = contenido;
   }
}
```

- "MensajeChat.java":

```
package estadocliente;
import java.io.Serializable;
public class MensajeChat implements Serializable {
   private String nombreAutor;
   private String fechaInstante;
   private String contenido;
   public MensajeChat(){}
   public String getNombreAutor() {
      return nombreAutor;
   }
   public void setNombreAutor(String nombreAutor) {
      this.nombreAutor = nombreAutor;
   }
   public String getFechaInstante() {
      return fechaInstante;
```

```
    }
    public void setFechaInstante(String fechaInstante) {
        this.fechaInstante = fechaInstante;
    }
    public String getContenido() {
        return contenido;
    }
    public void setContenido(String contenido) {
        this.contenido = contenido;
    }
}
```

Necesitamos una clase que genere los objetos que se envían al cliente: "MensajeFlash" y "EstadoCliente". Dicha clase es la que da servicio al cliente y está compuesta por métodos estáticos. Se encuentra en el paquete "serviciocliente" y se llama *ServicioCliente*.

- Clase "ServicioCliente.java":

```
package serviciocliente;
import domainmodel.BlogPost;
import domainmodel.ComentarioPost;
import estadocliente.Comentario;
import estadocliente.EstadoCliente;
import estadocliente.MensajeChat;
import estadocliente.MensajeFlash;
import estadocliente.Post;
import estadocliente.Usuario;
import exceptions.BlogException;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import service.BlogServiceRemote;
public class ServicioCliente {
    private static BlogServiceRemote blogService = null;
    public static EstadoCliente obtenerEstadoCliente(String nombreUsuario)
            throws BlogException {
        EstadoCliente estadoCliente = null;
        try {
            if (blogService == null) {
                blogService = obtenerReferenciaBlogService();
            }
            estadoCliente = new EstadoCliente();
            estadoCliente.setNombreUsuario(nombreUsuario);
            estadoCliente.setUsuarios(getUsuariosRelacionadosUsuario(nombreUsuario));
```

```
            estadoCliente.setPosts(getTodosLosPosts());
            String chatActivoCon = getNombreUsuarioChatActivo(nombreUsuario);
            estadoCliente.setChatActivo(chatActivoCon);
            if (chatActivoCon.equals("")) {
               estadoCliente.setChats(new ArrayList<MensajeChat>());
            } else {
               estadoCliente.
                     setChats(getMensajesChatActivo(nombreUsuario,chatActivoCon));
            }
         } catch(NamingException | BlogException e) {
            if (e instanceof BlogException) {
               throw new BlogException(e.getMessage());
            } else if (e instanceof NamingException){
               throw new BlogException("Error interno");
            }
         }
         return estadoCliente;
      }
      public static MensajeFlash obtenerMensajeFlash(String nombreUsuario,
            String mensaje, String error) {
         MensajeFlash mensajeF = new MensajeFlash();
         mensajeF.setNombreUsuario(nombreUsuario);
         mensajeF.setTitulo("El mejor blog del mundo");
         mensajeF.setMensaje(mensaje);
         mensajeF.setError(error);
         return mensajeF;
      }
      private static BlogServiceRemote obtenerReferenciaBlogService()
            throws NamingException {
         Hashtable hash = new Hashtable();
         hash.put(Context.INITIAL_CONTEXT_FACTORY,
               "weblogic.jndi.WLInitialContextFactory");
         hash.put(Context.PROVIDER_URL, "t3://localhost:7001");
         Context ctx = new InitialContext(hash);
         BlogServiceRemote bsr = (BlogServiceRemote)
               ctx.lookup("java:global.Blog.Blog-ejb.BlogService!"
                     + "service.BlogServiceRemote");
         return bsr;
      }
      private static List<Usuario> getUsuariosRelacionadosUsuario
            (String nombreUsuario) throws BlogException {
         List<Usuario> usuariosSistema = new ArrayList<>();
         List<String> nombresUsuarios = blogService.obtenerTodosUsuarios();
         String chatActivo = blogService.obtenerChatActualUsuario(nombreUsuario);
         for(String esteUsuario : nombresUsuarios) {
            Usuario u = new Usuario();
```

```
            u.setNombre(esteUsuario);
            if (esteUsuario.equals(nombreUsuario)) {
                u.setYoMismo(true);
            }
            if (esteUsuario.equals(chatActivo)) {
                u.setChateando(true);
            }
            usuariosSistema.add(u);
        }
        return usuariosSistema;
    }
    private static List<Post> getTodosLosPosts() throws BlogException {
        List<BlogPost> postsDominio = blogService.obtenerTodosLosPosts();
        List<Post> postsSistema = new ArrayList<>();
        for (BlogPost estePost : postsDominio) {
            Post p = new Post();
            p.setAutor(estePost.getAutor());
            p.setFechaInstante(estePost.getFechaInstante());
            p.setContenido(estePost.getContenido());
            p.setId(estePost.getId().toString());
            List<ComentarioPost> comentariosPost = estePost.getComentarios();
            List<Comentario> cPost = new ArrayList<>();
            for(ComentarioPost esteComentario : comentariosPost) {
                Comentario c = new Comentario();
                c.setContenido(esteComentario.getComentario());
                c.setFechaInstante(esteComentario.getFechaInstante());
                c.setNombreUsuario(esteComentario.getUsuario());
                cPost.add(c);
            }
            p.setComentarios(cPost);
            postsSistema.add(p);
        }
        return postsSistema;
    }
    private static String getNombreUsuarioChatActivo(String nombreUsuario)
            throws BlogException {
        String usuarioChatActivo = blogService.obtenerChatActualUsuario(nombreUsuario);
        return usuarioChatActivo; // Contiene el nombre o una cadena vacía.
    }
    private static List<MensajeChat> getMensajesChatActivo(String nombreUsuario1,
            String nombreUsuario2) throws BlogException {
        List<domainmodel.MensajeChat> mensajes =
                blogService.obtenerTodosLosMensajesChat(nombreUsuario1, nombreUsuario2);
        List<MensajeChat> mensajesUsuarios = new ArrayList<>();
        for(domainmodel.MensajeChat esteMensaje : mensajes) {
            MensajeChat chat = new MensajeChat();
```

```
            chat.setNombreAutor(esteMensaje.getEnviadoPor());
            chat.setContenido(esteMensaje.getContenido());
            chat.setFechaInstante(esteMensaje.getFechaInstante());
            mensajesUsuarios.add(chat);
        }
        return mensajesUsuarios;
    }
}
```

Dicha clase representa la generación de la información que se envía al cliente, pero ¿y la información que se recibe desde éste? Dicha información debe ser recibida y mapeada por una clase Java. Como vimos en el planteamiento del ejercicio, la información que se envía del cliente al servidor es un objeto JSON de la siguiente forma:

```
var operacion = {
        definicion: "",
        nombreUsuario: "",
        idPost: "",
        contenidoTexto: "",
        usuarioChat: ""
    };
```

Donde *definición* puede tomar alguno de los siguientes valores:

- keepalive.
- login.
- escribePost.
- escribeComentario.
- chateacon.
- enviachat.

La clase Java que mapea dicha información es la siguiente:

- Clase "operacion.java":

```
package socket.modelo;
import java.io.Serializable;
public class Operacion implements Serializable{
   private String definicion;
   private String nombreUsuario;
   private String idPost;
   private String contenidoTexto;
   private String usuarioChat;
   public Operacion() {}
   public String getDefinicion() {
      return definicion;
```

```
    }
    public void setDefinicion(String definicion) {
        this.definicion = definicion;
    }
    public String getNombreUsuario() {
        return nombreUsuario;
    }
    public void setNombreUsuario(String nombreUsuario) {
        this.nombreUsuario = nombreUsuario;
    }
    public String getIdPost() {
        return idPost;
    }
    public void setIdPost(String idPost) {
        this.idPost = idPost;
    }
    public String getContenidoTexto() {
        return contenidoTexto;
    }
    public void setContenidoTexto(String contenidoTexto) {
        this.contenidoTexto = contenidoTexto;
    }
    public String getUsuarioChat() {
        return usuarioChat;
    }
    public void setUsuarioChat(String usuarioChat) {
        this.usuarioChat = usuarioChat;
    }
}
```

La clase "BlogServlet" que se muestra en la estructura de archivos no contiene información relevante. De hecho, dicho *servlet* no se usa y podría eliminarse.

Ya sólo nos queda ver cómo encajan todas las piezas del puzle en el lado del cliente y en el lado del servidor.

En el lado del cliente tenemos:

- La última versión de la librería jQuery.
- La aplicación que se ejecuta cuando se entra o se sale del blog (cuando se recarga la página JSF): "blog.js".
- La aplicación que se ejecuta una vez dentro del blog, y que maneja los eventos del usuario y actualiza el estado de la página: "VistaDOM.js".

Veamos el código:

- "JavaScript blog.js":

```
var vistaDOM = null;
var nombreUsuario = null;
$(document).ready(function(){
  vistaDOM = new VistaDOM();
  vistaDOM.mensajeFlash();
  this.nombreUsuario = vistaDOM.obtenerNombreUsuario();
  if (this.nombreUsuario !== '') vistaDOM.setWebSocket(this.nombreUsuario);
});
```

El corazón del cliente del blog es "VistaDOM.js".

- JavaScript "VistaDOM.js":

```
var VistaDOM = function () {
  this.webSocket = null;
  this.nombreUsuario = null;
};
VistaDOM.prototype.obtenerNombreUsuario = function () {
  var $divNombreUsuario = $('div#nombreusuario');
  var nombreUsuario = $divNombreUsuario.html();
  var nombreUsuarioSinEspacios = nombreUsuario.trim();
  if (nombreUsuarioSinEspacios !== '') {
    nombreUsuario = nombreUsuario.trim();
    this.nombreUsuario = nombreUsuario;
    return nombreUsuario;
  } else {
    return '';
  }
};
VistaDOM.prototype.setWebSocket = function (nombreUsuario) {
  var sThis = this;
  this.asociaEventos();
  this.webSocket = new WebSocket("ws://localhost:7001/Blog-war/blogsocket");

  setTimeout(function () {
    sThis.webSocket.onmessage = function (mensaje) {
      sThis.recibeMensaje(mensaje);
    };
    sThis.setKeepAlive();
    // Espera para enviar el nombre de usuario.
    var operacion = {
      definicion: "login",
      nombreUsuario: nombreUsuario,
      idPost: "",
```

```
            contenidoTexto: "",
            usuarioChat: ""
        };
        var mensaje = JSON.stringify(operacion);
        sThis.webSocket.send(mensaje);
    }, 2000);
};
VistaDOM.prototype.setKeepAlive = function () {
    var sThis = this;
    setInterval(function () {
        var operacion = {
            definicion: "keepalive",
            nombreUsuario: sThis.nombreUsuario,
            idPost: "",
            contenidoTexto: "",
            usuarioChat: ""
        };
        var mensaje = JSON.stringify(operacion);
        sThis.webSocket.send(mensaje);
    }, 4000);
};
VistaDOM.prototype.recibeMensaje = function (mensaje) {
    var mensajeJson = $.parseJSON(mensaje.data);
    if (mensajeJson.operacion === 'mensajeflash') {
        this.seteaDatosFlash(mensajeJson);
        this.mensajeFlash();
    } else if (mensajeJson.operacion === 'actualiza') {
        this.actualizaModelo(mensajeJson);
    }
};
VistaDOM.prototype.seteaDatosFlash = function(datos) {
    var $divMensajeFlash = $('div#mensajeflash');
    var $divMensaje = $divMensajeFlash.find('div.mensaje');
    var $divMensajeError = $divMensajeFlash.find('div.error');
    $divMensaje.empty();
    $divMensajeError.empty();
    var mensajeInfo = datos.mensaje;
    var mensajeError = datos.error;
    var mensajeInfoSinEspacios = mensajeInfo.trim();
    var mensajeErrorSinEspacios = mensajeError.trim();
    if (mensajeInfoSinEspacios !== '') {
        $divMensaje.html(mensajeInfo);
    }
    if (mensajeErrorSinEspacios !== '') {
        $divMensajeError.html(mensajeError);
    }
};
```

```
VistaDOM.prototype.mensajeFlash = function () {
   var $divMensajeFlash = $('div#mensajeflash');
   $divMensajeFlash.css('display','block');
   var $divMensaje = $divMensajeFlash.find('div.mensaje');
   var $divMensajeError = $divMensajeFlash.find('div.error');
   var contenidoMensaje = $divMensaje.html();
   var contenidoMensajeSinEspacios = contenidoMensaje.trim();
   var contenidoMensajeError = $divMensajeError.html();
   var contenidoMensajeErrorSinEspacios = contenidoMensajeError.trim();
   if (contenidoMensajeSinEspacios === '') {
      $divMensaje.removeClass('visible').addClass('oculto');
   } else {
      $divMensaje.removeClass('oculto').addClass('visible');
   }
   if (contenidoMensajeErrorSinEspacios === '') {
      $divMensajeError.removeClass('visible').addClass('oculto');
   } else {
      $divMensajeError.removeClass('oculto').addClass('visible');
   }
   setTimeout(function () {
      $divMensajeFlash.fadeOut("slow", function () {});
   }, 4000);
};
VistaDOM.prototype.actualizaModelo = function (estado) {
   this.actualizaModeloUsuario(estado.nombreUsuario);
   this.actualizaModeloUsuarios(estado.usuarios);
   this.actualizaModeloPosts(estado.posts);
   this.actualizaModeloChatActivo(estado.chatActivo);
   this.actualizaModeloMensajes(estado.chats);
   this.desasociaEventos();
   this.asociaEventos();
};
VistaDOM.prototype.desasociaEventos = function () {
   $('a').unbind('click');
   $('textarea#campochat').unbind('keypress');
};
VistaDOM.prototype.asociaEventos = function () {
   var sThis = this;
   $('a.escribepost').on('click', function (event) {
      event.preventDefault();
      var $textArea = $('div.escribepost textarea');
      if ($textArea.length > 0) {
         var contenidoPost = $textArea.val();
         $textArea.val('');
         var contenidoSinEspacios = contenidoPost.replace(' ', '');
         if (contenidoSinEspacios !== '') {
            var operacion = {
```

```
                definicion: "escribePost",
                nombreUsuario: sThis.nombreUsuario,
                idPost: "",
                contenidoTexto: contenidoPost,
                usuarioChat: ""
            };
            var mensaje = JSON.stringify(operacion);
            sThis.webSocket.send(mensaje);
        }
    }
});
$('a.envcomentario').on('click', function (event) {
    event.preventDefault();
    var $this = $(this);
    var idPostComentario = $this.attr('idpost');
    var $textArea = $('div.escribecomentariopost textarea[idpost="'
        + idPostComentario + '"]');
    if ($textArea.length > 0) {
        var contenidoComentario = $textArea.val();
        $textArea.val('');
        var contenidoSinEspacios = contenidoComentario.replace(' ', '');
        if (contenidoSinEspacios !== '') {
            var operacion = {
                definicion: "escribeComentario",
                nombreUsuario: sThis.nombreUsuario,
                idPost: idPostComentario,
                contenidoTexto: contenidoComentario,
                usuarioChat: ""
            };
            var mensaje = JSON.stringify(operacion);
            sThis.webSocket.send(mensaje);
        }
    }
});
$('a.chatear').on('click', function (event) {
    event.preventDefault();
    var $enlace = $(event.target);
    var nombreUsuarioChat = $enlace.attr('nombreusuario');
    var operacion = {
        definicion: "chateacon",
        nombreUsuario: sThis.nombreUsuario,
        idPost: "",
        contenidoTexto: "",
        usuarioChat: nombreUsuarioChat
    };
    var mensaje = JSON.stringify(operacion);
    sThis.webSocket.send(mensaje);
});
```

```
  $('a#enviarchat').on('click', function (event) {
    event.preventDefault();
    var $contenidoMensaje = $('div#enviochat textarea');
    var contenidoMensaje = $contenidoMensaje.val();
    $contenidoMensaje.val('');
    var contenidoSinEspacios = contenidoMensaje.replace(' ', '');
    if (contenidoSinEspacios !== '') {
      var $spanUsuarioChat = $('span#usuariochat');
      var nombreUsuarioChat = $spanUsuarioChat.attr('nombreusuario');
      var operacion = {
        definicion: "enviachat",
        nombreUsuario: sThis.nombreUsuario,
        idPost: "",
        contenidoTexto: contenidoMensaje,
        usuarioChat: nombreUsuarioChat
      };
      var mensaje = JSON.stringify(operacion);
      sThis.webSocket.send(mensaje);
    }
  });
  $('div#enviochat textarea').on('keypress', function (event) {
    if (event.which === 13) {
      event.preventDefault();
      var $contenidoMensaje = $('div#enviochat textarea');
      var contenidoMensaje = $contenidoMensaje.val();
      $contenidoMensaje.val('');
      var contenidoSinEspacios = contenidoMensaje.replace(' ', '');
      if (contenidoSinEspacios !== '') {
        var $spanUsuarioChat = $('span#usuariochat');
        var nombreUsuarioChat = $spanUsuarioChat.attr('nombreusuario');
        var operacion = {
          definicion: "enviachat",
          nombreUsuario: sThis.nombreUsuario,
          idPost: "",
          contenidoTexto: contenidoMensaje,
          usuarioChat: nombreUsuarioChat
        };
        var mensaje = JSON.stringify(operacion);
        sThis.webSocket.send(mensaje);
      }
    }
  });
};
VistaDOM.prototype.actualizaModeloUsuario = function (nombreUsuario) {
  if ((nombreUsuario) && (typeof (nombreUsuario) === 'string')
      && (nombreUsuario.length > 0)) {
```

```
        var $divNombreUsuario = $('div#nombreusuario');
        $divNombreUsuario.empty();
        $divNombreUsuario.html(nombreUsuario);
    }
};
VistaDOM.prototype.actualizaModeloUsuarios = function (listadoUsuarios) {
    var $divListadoUsuarios = $('div#listadousuarios');
    $divListadoUsuarios.empty();
    var nUsuarios = listadoUsuarios.length;
    for (var i = 0; i < nUsuarios; i++) {
        var esteUsuario = listadoUsuarios[i];
        var soyYo = esteUsuario.yoMismo;
        var chateando = esteUsuario.chateando;
        var nombre = esteUsuario.nombre;
        var $usuario = $('<div class="usuario"></div>');
        var $saltoLinea = $('<br />');
        $usuario.append(nombre);
        $usuario.append($saltoLinea);
        if ((!soyYo) && (!chateando)) {
            var $botonChatear = $('<a class="chatear button gray" nombreusuario="'
                + nombre + '" href="#">Chatear</a>');
            $usuario.append($botonChatear);
        }
        $divListadoUsuarios.append($usuario);
    }
};
VistaDOM.prototype.actualizaModeloPosts = function (listadoPosts) {
    var $areaPost = $('div.escribepost textarea');
    $areaPost.val('');
    if ((listadoPosts) && (listadoPosts instanceof Array)) {
        var $listadoPostsDOM = $('div#listadoposts');
        var $divsPostsCabecera = $listadoPostsDOM
            .find('div.tituloposts.cabecera');
        var $divsPosts = $listadoPostsDOM.find('div.post');
        $divsPostsCabecera.remove();
        $divsPosts.remove();
        var nPosts = listadoPosts.length;
        for (var i = 0; i < nPosts; i++) {
            var estePost = listadoPosts[i];
            var idPost = estePost.id;
            var fechaPost = estePost.fechaInstante;
            var nombreUsuario = estePost.autor;
            var contenido = estePost.contenido;
            var comentarios = estePost.comentarios;
            var $tituloPostDOM = $('<div class="tituloposts cabecera">Post</div>');
            var $postDOM = $('<div class="post"></div>');
```

```
        var $tituloUsuario = $('<b>Usuario: </b>');
        var $tituloFecha = $('<b>Fecha: </b>');
        var $tituloContenido = $('<b>Contenido: </b>');
        var $divNombreUsuario = $('<div></div>')
              .append($tituloUsuario).append(nombreUsuario);
        var $divFecha = $('<div></div>')
              .append($tituloFecha).append(fechaPost);
        var $divContenido = $('<div></div>')
              .append($tituloContenido).append(contenido);
        var $divEscribeComentario = $('<div class="escribecomentariopost"></div>');
        var $escribeTuComentario = $('<div>Escribe tu comentario</div>');
        var $textareaComentario = $('<textarea id="escribecomentario" idpost="'
              + idPost + '" name="escribecomentario"></textarea>');
        var $enlaceEnviaComentario =
              $('<a href="#" class="envcomentario button gray" idpost="'
                    + idPost + '">Enviar</a>');
        $divEscribeComentario.append($escribeTuComentario);
        $divEscribeComentario.append($textareaComentario);
        $divEscribeComentario.append($enlaceEnviaComentario);
        $postDOM.append($divNombreUsuario);
        $postDOM.append($divFecha);
        $postDOM.append($divContenido);
        $postDOM.append($divEscribeComentario);
        var nComentarios = comentarios.length;
        for (var j = 0; j < nComentarios; j++) {
          var esteComentario = comentarios[j];
          var nombreUsuarioComentario = esteComentario.nombreUsuario;
          var fechaComentario = esteComentario.fechaInstante;
          var contenidoComentario = esteComentario.contenido;
          var $divUsuario = $('<div><b>Usuario: </b></div>');
          $divUsuario.append(nombreUsuarioComentario);
          var $divFechaComentario = $('<div><b>Fecha: </b></div>');
          $divFechaComentario.append(fechaComentario);
          var $divContenidoComentario = $('<div><b>Comentario: </b></div>');
          $divContenidoComentario.append(contenidoComentario);
          var $divComentario = $('<div class="comentariopost"></div>');
          $divComentario.append($divUsuario);
          $divComentario.append($divFechaComentario);
          $divComentario.append($divContenidoComentario);
          $postDOM.append($divComentario);
        }
        $listadoPostsDOM.append($tituloPostDOM);
        $listadoPostsDOM.append($postDOM);
      }
    }
};
```

```
VistaDOM.prototype.actualizaModeloChatActivo = function (chatActivo) {
   if ((chatActivo) && (typeof (chatActivo) === 'string') && (chatActivo.length > 0)) {
      var nombreUsuarioChat = chatActivo;
      var $divTituloChat = $('div#enviochat>div.titulo');
      $divTituloChat.empty();
      if (nombreUsuarioChat !== '') {
         $divTituloChat.append('Chatea con ' + nombreUsuarioChat);
      } else {
         $divTituloChat.append('Espacio para el chat');
      }
      var $spanUsuarioChat = $('span#usuariochat');
      $spanUsuarioChat.attr('nombreusuario', chatActivo);
   }
};
VistaDOM.prototype.actualizaModeloMensajes = function (mensajesChatActivo) {
   if ((mensajesChatActivo) && (mensajesChatActivo instanceof Array)) {
      var $divContenedorChats = $('div#contenedorchats');
      $divContenedorChats.empty();
      var nMensajes = mensajesChatActivo.length;
      for (var i = 0; i < nMensajes; i++) {
         var esteMensaje = mensajesChatActivo[i];
         var nombreUsuario = esteMensaje.nombreAutor + ': ';
         var contenido = esteMensaje.contenido;
         var $divMensaje = $('<div class="mensaje"></div>');
         var $usuario = $('<b></b>');
         $usuario.append(nombreUsuario);
         $divMensaje.append($usuario);
         $divMensaje.append(contenido);
         $divContenedorChats.append($divMensaje);
      }
   }
};
```

Y sólo nos queda por ver el corazón de la aplicación en el servidor.

- Clase "BlogSocket.java":

```
package socket;
import domainmodel.BlogPost;
import domainmodel.ChateaCon;
import domainmodel.ComentarioPost;
import domainmodel.MensajeChat;
import estadocliente.EstadoCliente;
import estadocliente.MensajeFlash;
import exceptions.BlogException;
import java.io.IOException;
import java.util.Collection;
```

```
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import service.BlogServiceRemote;
import socket.modelo.Operacion;
import util.JSONManager;
import serviciocliente.ServicioCliente;
@ServerEndpoint("/blogsocket")
public class BlogSocket {
   private BlogServiceRemote bsr = null;
   private static Map<String, Session> usuarios = new HashMap<String, Session>();
   @OnOpen
   public void abrirSocket(Session sesionUsuario) {
      usuarios.put(sesionUsuario.getId(), sesionUsuario);
   }
   @OnMessage
   public void recibeMensaje(String mensaje, Session sesionUsuario)
         throws IOException {
      synchronized (this) {
         String nombreUsuario = "";
         try {
            if (bsr == null) {
               bsr = obtenerReferenciaBlogService();
            }
            Operacion operacion = (Operacion) JSONManager
                  .generateTOfromJson(mensaje, Operacion.class);
            if (operacion.getDefinicion().equals("login")) {
               nombreUsuario = operacion.getNombreUsuario();
               sesionUsuario.getUserProperties()
                     .put("nombreUsuario", nombreUsuario);
            } else if (operacion.getDefinicion().equals("escribePost")) {
               nombreUsuario = operacion.getNombreUsuario();
               String contenidoPost = operacion.getContenidoTexto();
               BlogPost post = new BlogPost();
               post.setAutor(nombreUsuario);
               post.setContenido(contenidoPost);
               bsr.crearPost(post);
```

```
        } else if (operacion.getDefinicion().equals("escribeComentario")) {
            nombreUsuario = operacion.getNombreUsuario();
            String idPost = operacion.getIdPost();
            String comentario = operacion.getContenidoTexto();
            BlogPost post = bsr.obtenerPostPorId(Long.valueOf(idPost));
            ComentarioPost comentarioPost = new ComentarioPost();
            comentarioPost.setComentario(comentario);
            comentarioPost.setUsuario(nombreUsuario);
            bsr.anadirComentarioPost(post, comentarioPost);
        } else if (operacion.getDefinicion().equals("chateacon")) {
            nombreUsuario = operacion.getNombreUsuario();
            String usuarioChateaCon = operacion.getUsuarioChat();
            ChateaCon chatea = new ChateaCon();
            chatea.setUsuario(nombreUsuario);
            chatea.setCon(usuarioChateaCon);
            bsr.abreChatCon(chatea);
        } else if (operacion.getDefinicion().equals("enviachat")) {
            nombreUsuario = operacion.getNombreUsuario();
            String usuarioChateaCon = operacion.getUsuarioChat();
            String mensajeCht = operacion.getContenidoTexto();
            MensajeChat mensajeChat = new MensajeChat();
            mensajeChat.setContenido(mensajeCht);
            mensajeChat.setEnviadoPor(nombreUsuario);
            mensajeChat.setEnviadoA(usuarioChateaCon);
            bsr.enviaMensajeChat(mensajeChat);
        }
        if (!operacion.getDefinicion().equals("keepalive")) {
            Collection<Session> sesiones = usuarios.values();
            Iterator it = sesiones.iterator();
            while (it.hasNext()) {
                Session estaSesion = (Session) it.next();
                String nombreEsteUsuario = (String) estaSesion
                        .getUserProperties().get("nombreUsuario");
                EstadoCliente eCliente
                        = ServicioCliente.obtenerEstadoCliente(nombreEsteUsuario);
                String estado = JSONManager.generateJson(eCliente);
                estaSesion.getBasicRemote().sendText(estado);
            }
        }
    } catch (BlogException e) {
        MensajeFlash mensajeFlash = ServicioCliente
                .obtenerMensajeFlash(nombreUsuario, "", e.getMessage());
        String mensajeF = JSONManager.generateJson(mensajeFlash);
        sesionUsuario.getBasicRemote().sendText(mensajeF);
    } catch (NamingException e) {
        MensajeFlash mensajeFlash = ServicioCliente
```

```
                .obtenerMensajeFlash(nombreUsuario, "", "Error interno");
            String mensajeF = JSONManager.generateJson(mensajeFlash);
            sesionUsuario.getBasicRemote().sendText(mensajeF);
          }
        }
    }
    @OnClose
    public void cerrarSocket(Session sesionUsuario) {
        usuarios.remove(sesionUsuario.getId());
    }
    private BlogServiceRemote obtenerReferenciaBlogService() throws NamingException {
        Hashtable hash = new Hashtable();
        hash.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        hash.put(Context.PROVIDER_URL, "t3://localhost:7001");
        Context ctx = new InitialContext(hash);
        BlogServiceRemote bsr = (BlogServiceRemote)
            ctx.lookup("java:global.Blog.Blog-ejb.BlogService!"
            + "service.BlogServiceRemote");
        return bsr;
    }
}
```

Bibliografía y fuentes de información

- Keogh, Jim. J2EE: *Manual de referencia*. McGraw-Hill/Interamericana de España. ISBN 9788448139803.
- Gamma Erich, Helm, Richard, Johnson, Ralph y Vlissides, John. *Patrones de Diseño: elementos de software orientado a objetos reutilizables*. Addison-Wesley. ISBN 9788478290598.
- Gulabana, Sunil. *Developing RESTful Web Services with Jersey 2.0*. Packt Publishing. ISBN 9781783288298.
- Ceballos, Francisco Javier. Java. *Interfaces Gráficas y Aplicaciones para Internet*. Ra-Ma. ISBN 9788499645223.