palasi.com
IT and management

Programación para el Web con Java Tema 3

Dr. Vicent-Ramon Palasí Lallana
Universidad Francisco Gavidia
Mayo 2005.

Programa del tema 3

- 3.1. Motores de persistencia.
- 3.2. Instalación de MySQL.
- 3.3. Instalación de Hibernate.
- 3.4. El patrón DAO.
- 3.5. Conceptos básicos de Hibernate.
- 3.6. El lenguaje HQL.

Programa del tema 3

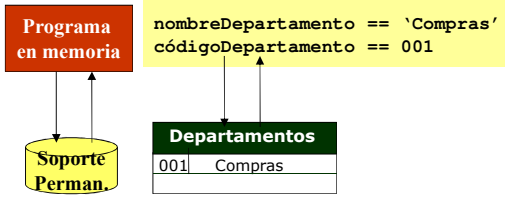
- 3.1. Motores de persistencia.
- 3.2. Instalación de MySQL.
- 3.3. Instalación de Hibernate.
- 3.4. El patrón DAO.
- 3.5. Conceptos básicos de Hibernate.
- 3.6. El lenguaje HQL.

Persistencia de datos

- Es la necesidad que tienen los programas de guardar los datos en un soporte permanente para que no se pierdan entre ejecuciones.
- Una vez guardados los datos hay que poder recuperarlos cuando se necesiten.
- Si se cumplen estas condiciones, se dice que el programa es persistente o que tiene persistencia de datos.

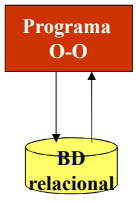
Persistencia de datos

- Cualquier aplicación comercial en cualquier lenguaje de programación debe tener persistencia de datos.
- Formas tradicionales de conseguirla: archivos, BDs.



Una situación típica

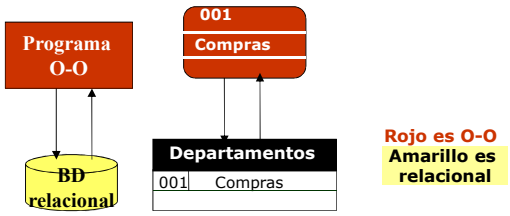
- La situación típica es aquella en que el programa es orientado a objetos (Java) y el soporte permanente es una base de datos relacional.
- El uso de una base de datos relacional para guardar los datos es el estándar desde hace 30 años.



palasi.com
IT and management

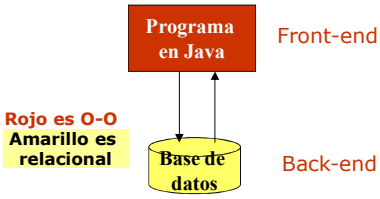
Un problema

- El programa trata los datos como objetos.
- La base de datos trata los datos como registros.
- Y sin embargo deben comunicarse.



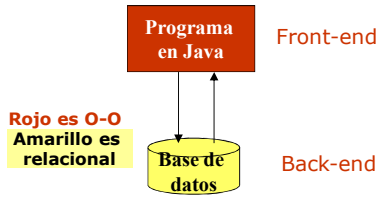
Es decir, tenemos dos estándares en una aplicación

- Para el programa: Modelo orientado a objetos.
- Para la base de datos. Modelo relacional.



¿Cómo se relacionan estos estándares?

- Son bastante diferentes, tanto en objetivos como en características de cada uno de ellos.



Objetivos

- **Modelo orientado a objetos**: Modelar conjuntamente los **datos y los procesos** de una aplicación.
- **Modelo relacional**: Modelar el almacenamiento y recuperación de **datos** de una aplicación.

Diferencias entre ambos modelos

- El modelo orientado a objetos considera los procesos mientras el relacional no.
- Pero incluso en la forma de tratar los datos ambos modelos difieren.
- Por ejemplo, uno tiene estructura y el otro no.

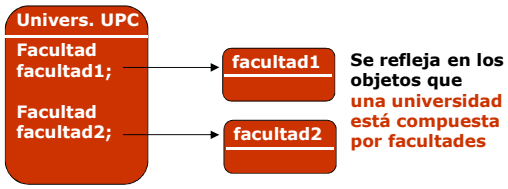
La realidad tiene una estructura

- Una universidad está compuesta de Facultades, cada una de las cuales está compuesta de Cursos y así sucesivamente.

Mail: palasi@palasi.com Web: www.palasi.com

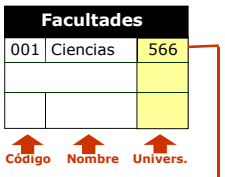
El modelo O-O refleja estructura de la realidad explícitamente

- Una universidad está compuesta de Facultades, cada una de las cuales está compuesta de Cursos y así sucesivamente.

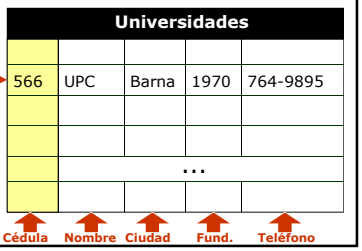


Se refleja en los objetos que una universidad está compuesta por facultades

El modelo relacional es relativamente plano



La estructura de la realidad está en forma implícita y debe ser el programador la que la tenga en cuenta.



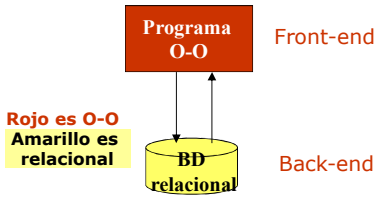
En este ejemplo, no se refleja en las tablas que una universidad está compuesta por facultades

Otras diferencias entre modelo O-O y relacional

- Uno trata con objetos y otro con registros.
- Uno tiene herencia y otro no.
- Uno representa las relaciones como atributos, el otro como claves foráneas.

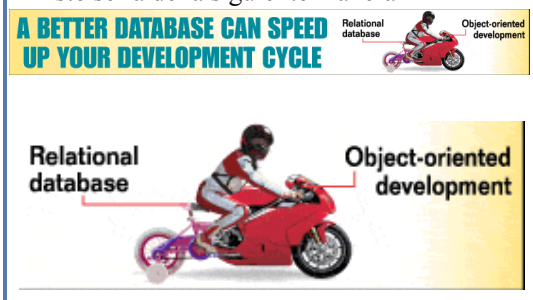
Tenemos dos estándares (dos idiomas) en una aplicación

- El programa trata los datos como orientado a objetos (“habla inglés”).
- La base de datos trata los datos de forma relacional (“habla alemán”).
- Pero tanto el programa como la BD deben intercambiar esos datos (“deben comunicarse”).



Visto desde la propaganda de Caché (una BD O-O)

- Esto sería de la siguiente manera



¿Cómo lo hacemos?

- Hay tres soluciones posibles:
- 1. Que el programa trabaje de forma relacional.
- 2. Que la base de datos trabaje O-O.
- 3. Usar un motor de persistencia

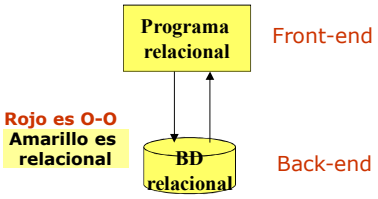
Mail: palasi@palasi.com Web: www.palasi.com

¿Cómo lo hacemos?

- Hay tres soluciones posibles:
- 1. Que el programa trabaje de forma relacional.
- 2. Que la base de datos trabaje O-O.
- 3. Usar un motor de persistencia

El programa trabaja con registros y no objetos, de forma relacional

- Esta es la forma tradicional ("ambos hablan alemán").
- Sin embargo, se pierden todas las ventajas de la orientación a objetos: reusabilidad, flexibilidad, facilidad de mantenimiento.
- No parece lógico renunciar a la orientación a objetos siendo el estándar actual y además programando en Java.



¿Cómo lo hacemos?

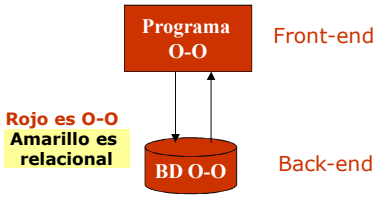
- Hay tres soluciones posibles:
- 1. Que el programa trabaje de forma relacional.
- 2. Que la base de datos trabaje O-O.
- 3. Usar un motor de persistencia

Bases de datos orientadas a objetos

- Son bases de datos que guardan objetos y devuelven objetos (en vez de registros).
- Caché y algunas más.

Ahora ambos componentes trabajan con objetos

- Esta es la mejor opción sobre el papel ("todos hablan inglés").
- Trabajamos sobre el estándar actual de orientación a objetos.
- La BD guarda objetos y el programa recupera objetos desde ella.



Sin embargo, en la práctica esta solución no es tan maravillosa

- La tecnología de BDs O-O no está madura.
- Las mejores BDs son relacionales.
- La inmensa mayoría de herramientas existe para BD relacionales: nos limitamos mucho.
- No hay un estándar para BDs O-O: nos ligamos a un vendedor específico ("vendor lock-in"), con el riesgo de que desaparezca, dé un mal servicio o suba los precios de forma abusiva.

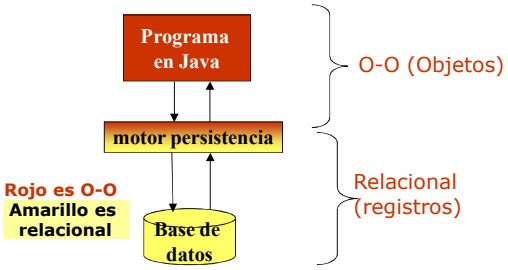
Mail: palasi@palasi.com Web: www.palasi.com

¿Cómo lo hacemos?

- Hay tres soluciones posibles:
- 1. Que el programa trabaje de forma relacional.
- 2. Que la base de datos trabaje O-O.
- 3. Usar un motor de persistencia

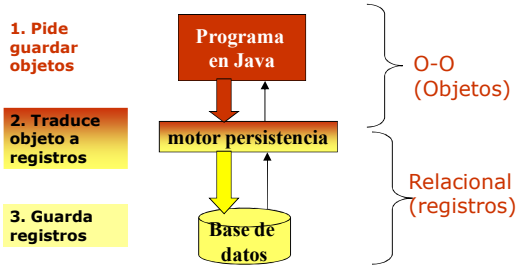
Motor de persistencia

- Hace la traducción entre el modelo O-O y relacional (y viceversa). ("Traductor entre inglés y alemán")



Guardando un objeto

- El programa sólo ve que pide guardar un objeto y el objeto se guarda.

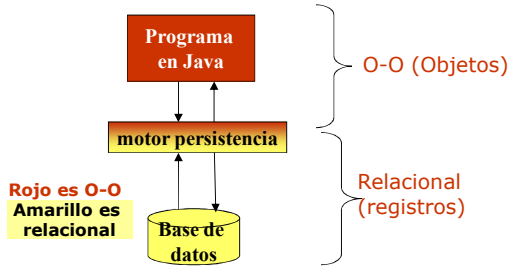


Recuperando un objeto

- El programa sólo ve que pide recuperar un objeto y el objeto se recuperar.

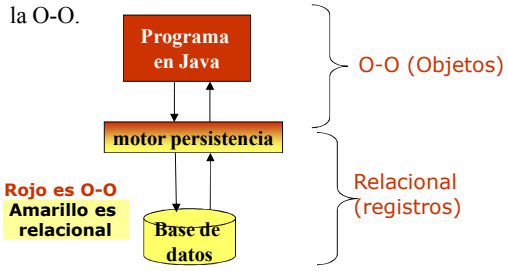
Motor de persistencia

- Es una capa (conjunto de clases) que hace la traducción de objetos a registros y viceversa).



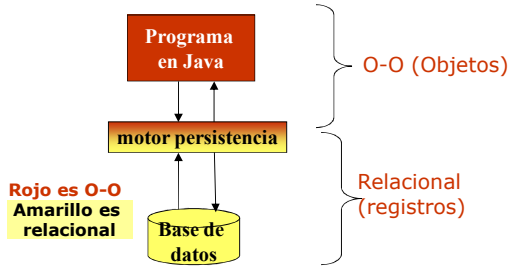
Motor de persistencia

- Así podemos tener lo mejor de los dos mundos: trabajamos con una BD relacional (con herramientas) y programamos con las ventajas de la O-O.



Motor de persistencia

- Puede añadir una cierta sobrecarga (normalmente pequeña) o, por el contrario, puede ser más eficiente porque puede implementar cachés.



Motor de persistencia

- Como vemos es la mejor opción: permite tener lo mejor de ambos mundos.
- ¿Y dónde encontramos esta maravilla?
- Hay dos opciones:
  - Programar una nosotros.
  - Usar una que ya esté programada.

¿Qué haremos?

- Programar **y mantener** una motor de persistencia es una tarea compleja y demanda recursos.
- Nuestro consejo es no invertir esfuerzo y dinero en esta tarea.
- Es mejor conseguir una motor de persistencia ya programada y trabajar con ella.

Motores de persistencia ya programados en el mundo Java

- Pueden ser comerciales (hay que pagar por ellos).
- O de código abierto y gratuitos.
- Hay mucha variedad. A continuación veremos algunos ejemplos.

Algunos motores de persistencia comerciales

- **Toplink (de Oracle)**. http://otn.oracle.com/products/ias/toplink/index.html
- **Kodo JDO** http://www.solarmetric.com/
- **FastObjects.** http://www.versant.com/products/fastobjects
- **Cocobase**. http://www.thoughtinc.com/
- **FrontierSuite**. http://www.objectfrontier.com/
- **Lido** http://www.xcalia.com/products/lido.jsp
- **IntelliBO** http://www.signsoft.com/en/intellibo/index.jsp
- **JCredo.** http://jcredo.com/home/index.jsp
- **VDBS O-R Framework.** http://www.objectmatter.com/

Algunos motores de persistencia de código abierto

- **Hibernate.** http://www.hibernate.org
- **Castor.** Soporta XML http://www.castor.org.
- **ObjectRelationalBridge**. Se integró en Jakarta http://db.apache.org/ojb/
- **Torque** http://db.apache.org/torque/
- **Cayenne**. http://sourceforge.net/projects/cayenne/
- **TriactiveJDO**. http://tjdo.sourceforge.net/
- **SpeedoJDO**. http://speedo.objectweb.org/
- **JGrinder**. Probada para grandes volúmenes http://sourceforge.net/projects/jgrinder/
- **Osage**. http://osage.sourceforge.net/
- **Tornado**. http://sourceforge.net/projects/tornado-db/

And the winner is…

- **Hibernate.** Es el motor de persistencia más extendido en Java.
- Es gratuito, de código abierto, de calidad y adecuado para gran tráfico de datos.
- Soporta 15 bases de datos de las más populares. Soporte a otras BDs es fácil de añadir.
- Buena documentación y foros de discusión.

Cultura general: ¿Qué es JDO?

- JDO (Java Data Objects) es un estándar para motores de persistencia.
- Si se adopta (ya se lleva años esperándolo), podríamos cambiar de motor sin cambiar el código.
- Sin embargo, JDO 1.0 es muy inmaduro y limitado por ahora para usarlo en aplicaciones comerciales.
- Se está comenzando a discutir JDO 2.0, que incorporará muchas de las ventajas que ya tiene Hibernate.

Programa del tema 3

Hibernate trabaja con estos SGBDs

- Oracle
- DB2
- SQL Server
- MySQL
- PostgreSQL
- Sybase
- SAP DB
- Informix
- Interbase
- HypersonicSQL
- FrontBase
- Ingres
- Progress
- Mckoi SQL
- Pointbase

Soporte a otras BDs es fácil de implementar

En Hibernate se trabaja igual con todos los SGBDs

- Para cambiar de SGBD (por ejemplo, de SQL Server a Oracle), basta con cambiar los archivos de configuración para indicar el SGBD.
- No hay que cambiar nada del programa.
- Esta es una de las ventajas de un motor de persistencia.

Sin embargo

- Algún SGBD debemos elegir para practicar en este curso.
- Elegiremos MySQL.

Mail: palasi@palasi.com Web: www.palasi.com

MySQL

- SGBD muy eficiente y de código abierto.
- Esto quiere decir que puede ser usado gratuitamente.

Ventajas de MySQL

- Adecuado para conjuntos de datos muy grandes. Permite un buen rendimiento incluso en tablas de millones de registros.
- Adecuado para acceso concurrente de cientos de usuarios al mismo tiempo.
- Se escala muy bien.
- Está disponible para un gran número de sistemas operativos y plataformas.

Inconvenientes de MySQL

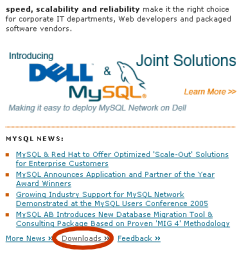
- No soporta completamente el estándar SQL (aunque casi ninguna base de datos lo hace).
- No soporta relaciones de integridad referencial ni procedimientos almacenados hasta la versión 5.

Instalación y configuración de MySQL

- Si es en Windows, debemos tener una cuenta que sea "Administrador del sistema".
- Si tenemos instalada una versión anterior de MySQL, debemos parar el servidor antes de instalar la nueva versión.

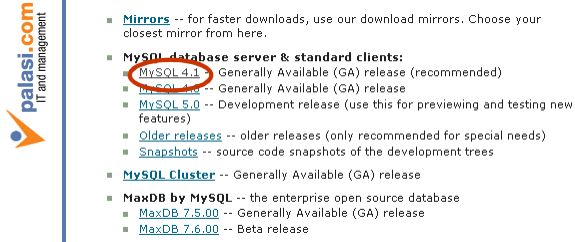
Instalación y configuración de MySQL

- En la página principal de MySQL, **http://www.mysql.com** hagan clic en **Downloads**



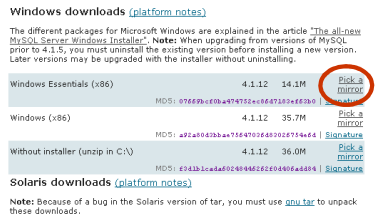
Instalación y configuración de MySQL

- En la página que aparece hacemos clic en la última versión de producción de MySQL.



Instalación y configuración de MySQL

- En la página que aparece, se va a la sección de **Windows downloads**, dentro de **Windows essentials** y se hace clic en **Pick a mirror**.

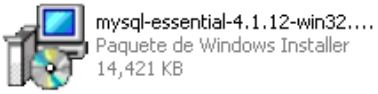


Instalación y configuración de MySQL

- En la página que aparece, se hace clic en **No thanks, just take me to the downloads!** Aparece una página con una lista de mirrors y se hace clic en uno de ellos.

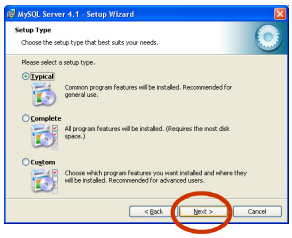
Instalación y configuración de MySQL

- Una vez descargado aparecerá un archivo de instalación que ejecutaremos.



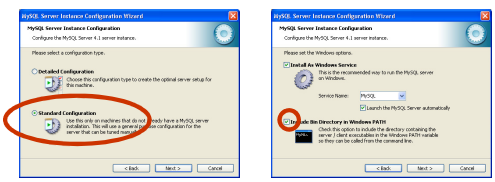
Instalación y configuración de MySQL

- Es un programa de instalación normal. Escogeremos la opción **Typical**. El programa de instalación se ejecutará normalmente. Escogeremos **Skip Signup.**



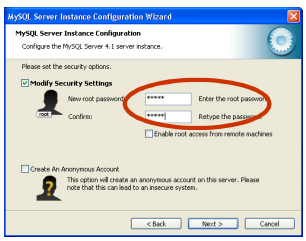
Elige estas opciones

- Todas las opciones son las predeterminadas excepto que elegimos **Standard Configuration** y, más tarde, **Include Bin Directory in Windows PATH**.



Elige estas opciones

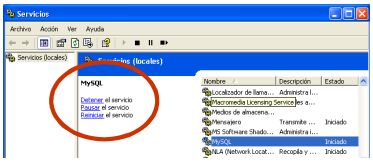
- Nos pedirá la contraseña del usuario **root**, que es el administrador de la base de datos.



Mail: palasi@palasi.com Web: www.palasi.com

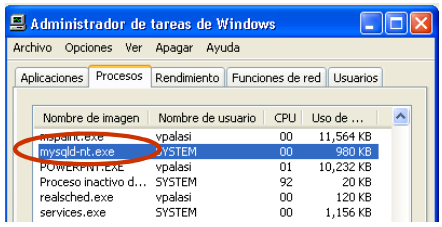
Se creará un servicio de Windows conteniendo el servidor MySQL

- El servicio se iniciará automáticamente cuando se inicie Windows.
- Podemos pararlo o reiniciarlo desde **Panel de control|Herramientas administrativas|Servicios|MySQL.**



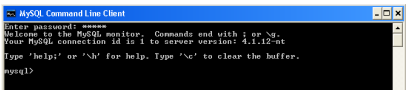
El servidor de MySQL estará iniciado

- Podemos verlo en la lista de procesos del sistema.



Para hacer operaciones con MySQL

- Podemos utilizar el cliente de mantenimiento predeterminado de la BD, que es de modo texto.
- Para iniciarlo, hacemos **Inicio|Todos los programas|MySQL|MySQL Server 4.1|MySQL Command Line Client**
- Nos pedirá la contraseña (pondremos la de root). Aparecerá un prompt para que introduzcamos comandos



Algunas operaciones del cliente de texto de MySQL

- `\h` Ayuda sobre el cliente SQL.
- `\q` Salir del cliente
- `\u` ***nombreBase*** Trabajar con la BD ***nombreBase***
- Cualquier comando SQL siempre que lo acabemos en punto y coma.

Hagan estas operaciones sobre el cliente

- `show databases;` Verán las bases de datos que hay.
- `\u test` Trabajaremos con la base de datos test.
- `show tables;` Verán las tablas de esas bases de datos.
- Creen una tabla "prueba" con

```
CREATE TABLE prueba (nombre CHAR(10));
show tables;
```

- Introduzcan un registro con

```
INSERT INTO prueba VALUES ("Juan");
```

- `select * from prueba;`
- `\q`

Más información

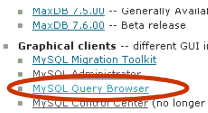
- MySQL Reference Manual.
- Se puede descargar de **http://dev.mysql.com/doc/**

Mail: palasi@palasi.com Web: www.palasi.com

Instalación de MySQL Query Browser

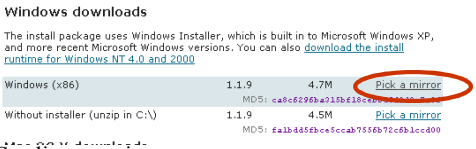
- MySQL Query Browser es un cliente gráfico para consultar y editar los datos de servidores MySQL.
- Se puede descargar del sitio:

http://dev.mysql.com/downloads/

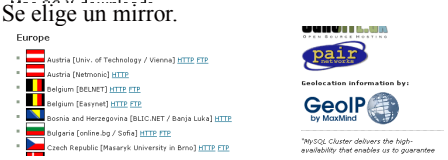


Instalación de MySQL Query Browser

- Se elige el enlace Pick a mirror en la parte de Windows downloads.



- Se elige un mirror.



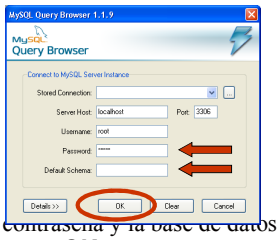
Instalación de MySQL Query Browser

- Aparece un programa de instalación.

- Se ejecuta el programa de instalación, que es típico. Se eligen todas las opciones predeterminadas.

Ejecución de MySQL Query Browser

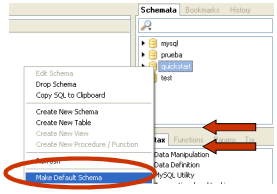
- Ejecutamos haciendo **Inicio|Todos los programas|MySQL|MySQL Query Browser**.



- Ponemos la contraseña y la base de datos por defecto. Hacemos **OK**.

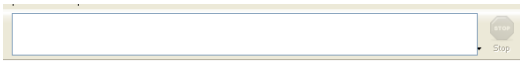
Ejecución de MySQL Query Browser

- Para usar una base de datos, se hace clic derecho en su nombre en el área llamada **Schemata** y se elige **Make default schema**.

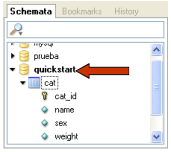


Ejecución de MySQL Query Browser

- En el cuadro de texto superior se pueden escribir consultas SQL. Para ejecutarlas, se hace Ctrl+Enter.



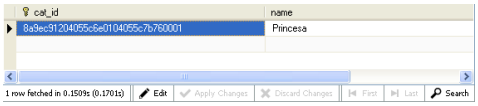
- Si se expande la base de datos en Schemata, aparece el nombre de las tablas. Haciendo doble clic en una tabla, aparece una consulta que devuelve los datos de esa tabla.



Mail: palasi@palasi.com Web: www.palasi.com

Ejecución de MySQL Query Browser

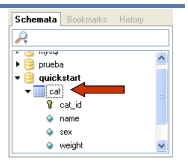
- En el resultado de una consulta, se hace clic en **Edit**.

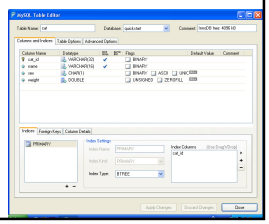


- Se pueden editar los campos, añadir y eliminar registros (haciendo clic derecho en el registro y seleccionando **Add row** o bien **Delete row**).
- Cuando acabemos de editar, debemos hacer clic en el botón **Apply Changes** para que los cambios se guarden.

Ejecución de MySQL Query Browser

- Haciendo clic derecho en el nombre de una tabla y seleccionando **Edit table** podemos cambiar la estructura de una tabla (campos, etc.).





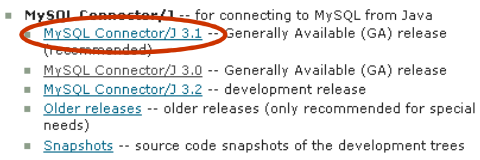
- Los campos que no tengan valor por defecto incluir Default Value como NULL en Column Details.

Controlador JDBC para MySQL

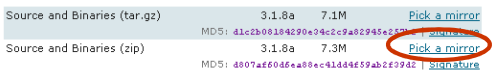
- Es la librería que utiliza Java (y, por tanto, Hibernate) para acceder a la BD MySQL
- Debemos descargarla de Internet.

Instalación del controlador JDBC para NySQL

- En la página http://dev.mysql.com/downloads/ se va a la versión recomendada de Connector/J (el nombre del controlador JDBC para MySQL).



Hacemos clic en Pick a mirror al lado de la versión del archivo ZIP



- Como resultado de la descarga, obtenemos un archivo ZIP.

Descomprimimos el archivo ZIP

- En el directorio "**`mysql-connector-java-3.1.8`**" (o similar) aparece un archivo JAR (**`mysql-connector-java-3.1.8-bin.jar`**, o similar). Ese es el controlador.

mysql-connector-java-3.1.8-bin.jar

Mail: palasi@palasi.com Web: www.palasi.com

Peculiaridades del controlador JDBC para MySQL

- La clase del controlador es
  `com.mysql.jdbc.Driver`
- La estructura de las URL de conexión es:
- `jdbc:mysql://nombreHost/nombreBD`
  donde ***nombreHost*** puede ser una dirección IP.
- Para la máquina local, podemos poner:
  `jdbc:mysql://localhost/nombreBD`

Programa del tema 3

- 3.1. Motores de persistencia.
- 3.2. Instalación de MySQL.
- 3.3. Instalación de Hibernate.
- 3.4. El patrón DAO.
- 3.5. Conceptos básicos de Hibernate.
- 3.6. El lenguaje HQL.

3.3. Instalación de Hibernate

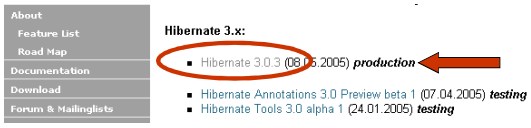
- 3.3.1. Descargar Hibernate.
- 3.3.2. Copiar JARs.
- 3.3.3. Colocar archivos de configuración.
- 3.3.4. Desplegar el ejemplo de prueba.

3.3. Instalación de Hibernate

- 3.3.1. Descargar Hibernate.
- 3.3.2. Copiar JARs.
- 3.3.3. Colocar archivos de configuración.
- 3.3.4. Desplegar el ejemplo de prueba.

Descargando Hibernate

- En la página de Hibernate www.hibernate.org se hace clic en Download. Aparece la siguiente página

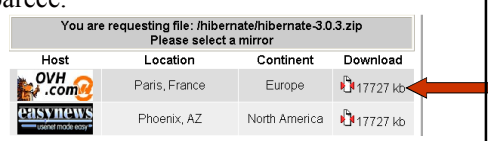


- Se hace clic en la versión de producción.

En la página que aparece

- Se hace clic en la versión recomendada con extensión zip.

- Se elige un mirror (espejo) en la página que aparece.



Se descargará un archivo ZIP

- Que contiene Hibernate.

3.3. Instalación de Hibernate

- 3.3.1. Descargar Hibernate.
- 3.3.2. Copiar JARs.
- 3.3.3. Colocar archivos de configuración.
- 3.3.4. Desplegar el ejemplo de prueba.

Se descomprime el archivo ZIP que se ha descargado

- Aparece el directorio de Hibernate, que pondremos en el lugar del disco duro.

- Nosotros le cambiaremos del nombre a hibernate y lo pondremos en la raíz. Por eso, será **`C:\hibernate`**. A este directorio lo llamaremos ***HIBERNATE***

Debemos copiar unos archivos JAR en dos directorios de Tomcat

- Un directorio es el **`common\lib`** de la instalación de Tomcat. Lo llamaremos el "directorio global".
- Otro es el directorio **`WEB-INF\lib`** del contexto. Lo llamamos "directorio local".
  - En el contexto por defecto, sería **`webapps\ROOT\WEB-INF\lib`**.
  - En otro contexto **`webapps\`*`nombreContexto`*`\WEB-INF\lib`**.

Copiar JARs

- El controlador JDBC de la base de datos (en nuestro caso, **`mysql-connector-java-3.1.8-bin.jar`**) se copiará en el directorio global.
- Recordemos que este controlador lo obteníamos descomprimiendo el ZIP que habíamos descargado.
- Este será el único archivo que copiaremos en el directorio global.

En el directorio local, copiaremos

- El archivo ***HIBERNATE*****`\hibernate3.jar`**
- Los archivos ***HIBERNATE*****`\lib\ehcache-1.1.jar`** y ***HIBERNATE*****`\lib\jta.jar`** (o similares)
- Todos los archivos JAR que hay en ***HIBERNATE*****`\lib`** y que son obligatorios para el runtime (excepto los que son sólo obligatorios para standalone).
- Sabremos cuáles son estos archivos: los veremos en ***HIBERNATE*****`\lib\_README.txt`**.

Mail: palasi@palasi.com Web: www.palasi.com

Archivos obligatorios para Hibernate 3.1.8

```
antlr-2.7.5H3.jar
asm.jar
asm-attrs.jar
cglib-2.1.jar
commons-collections-2.1.1.jar
commons-logging-1.0.4.jar
dom4j-1.6.jar
ehcache-1.1.jar
jta.jar
xerces-2.6.2.jar
xml-apis.jar
```

- Son los que pondremos en el directorio local junto a **`hibernate3.jar`**.

3.3. Instalación de Hibernate

- 3.3.1. Descargar Hibernate.
- 3.3.2. Copiar JARs.
- 3.3.3. Colocar archivos de configuración.
- 3.3.4. Desplegar el ejemplo de prueba.

Creamos un archivo con el contenido siguiente

```
<Context path="trayectoria"
  docBase="directorio">
<Resource name="jdbc/nombreJNDI"
  scope="Shareable"
  type="javax.sql.DataSource"
  factory="org.apache.tomcat.dbcp.dbcp.
  BasicDataSourceFactory"
  url="URLdelaBD"
  driverClassName="ClaseDelDriver"
  username="usernameBD"
  password="passwordBD" maxWait="3000"
  maxIdle="100" maxActive="10">
</Resource>
</Context>
```

En el anterior archivo

- *nombreJNDI:* Es un nombre arbitrario pero es bueno que coincida con el nombre de la BD.
- *trayectoria*: prefijo para el contexto precedido por /. Si es el contexto por defecto, es la cadena vacía.
- *directorio*: nombre del subdirectorio de **`webapps`** donde se encuentra el contexto. En el contexto por defecto es **`ROOT`**.
- *URLdelaBD*: URL para acceder a la BD. En MySQL es **`jdbc:mysql://localhost/`*`nombre`***, donde ***nombre*** es el nombre de la BD.
- *ClaseDelDriver*: es la clase del driver JDBC. En MySQL es **`com.mysql.jdbc.Driver`**.
- *usernameBD* y *passwordBD*: son el username y el password para acceder a la BD.

Por ejemplo, para el contexto ROOT, sería el siguiente

```
<Context path="" docBase="ROOT">
<Resource name="jdbc/quickstart"
  scope="Shareable"
  type="javax.sql.DataSource"
  factory="org.apache.tomcat.dbcp.dbcp.Bas
  icDataSourceFactory"
  url="jdbc:mysql://localhost/quickstart"
  driverClassName="com.mysql.jdbc.Driver"
  username="root" password="admin"
  maxWait="3000" maxIdle="100"
  maxActive="10">
</Resource>
</Context>
```

Nota importante

- Este archivo hay que copiarlo de los archivos adjuntos a esta presentación (están en el mismo directorio que la presentación).
- No se debe copiar directamente de la transparencia pues puede tener caracteres extraños que den errores al ejecutarse.

Mail: palasi@palasi.com Web: www.palasi.com

Este contenido

- Se guarda en un archivo colocado en **`webapps\`*nombrecontexto*`\META-INF\context.xml`**, donde ***nombrecontexto*** es el nombre del subdirectorio de **`webapps`** donde se encuentra la aplicación.
- Por ejemplo, en el contexto por defecto es **`webapps\ROOT\META-INF\context.xml`**

Ahora ponemos varios archivos de configuración en un directorio

- El directorio es **`WEB-INF\classes`** del contexto.
- Este directorio lo llamamos "directorio de clases".
- Allí colocaremos:
  - **`hibernate.cfg.xml`**
  - los archivos de correspondencia.

Ahora ponemos varios archivos de configuración en un directorio

- El directorio es **`WEB-INF\classes`** del contexto.
- Este directorio lo llamamos "directorio de clases".
- Allí colocaremos:
  - **`hibernate.cfg.xml`**
  - los archivos de correspondencia.

`hibernate.cfg.xml` (colocar en el directorio de clases)

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-
  configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.datasource">
         java:comp/env/jdbc/nombreJNDI </property>
        <property name="show_sql">false</property>
        <property name="dialect"> dialecto</property>
           archivos de correspondencia de cada clase persist.
    </session-factory>
</hibernate-configuration>
```

En el archivo anterior

- *nombreJNDI*: Es el nombre que hemos puesto en el archivo de configuración XML que hay en **`conf\Catalina\localhost`** (mirar antes).
- *dialecto*: Dialecto del SQL de la BD. Se encuentra en la documentación de Hibernate. Para MySQL es **`org.hibernate.dialect.MySQLDialect`**
- *archivos de correspondencia* : son un conjunto de líneas

```
<mapping resource="nombre1"/>
...
<mapping resource="nombreN"/>
```

donde ***nombre1***,..., ***nombreN*** son los nombres de los archivos de correspondencia. Hay un archivo por cada clase persistente.

Por ejemplo, para el ejemplo `Cat`

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-
  configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.datasource">
         java:comp/env/jdbc/quickstart</property>
        <property name="show_sql">false</property>
        <property name="dialect">
        org.hibernate.dialect.MySQLDialect</property>
     <mapping resource="Cat.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Nota importante

- Este archivo hay que copiarlo de los archivos adjuntos a esta presentación (están en el mismo directorio que la presentación).
- No se debe copiar directamente de la transparencia pues puede tener caracteres extraños que den errores al ejecutarse.

Ahora ponemos varios archivos de configuración en un directorio

- El directorio es **`WEB-INF\classes`** del contexto.
- Este directorio lo llamamos "directorio de clases".
- Allí colocaremos:
  - **`hibernate.cfg.xml`**
  - los archivos de correspondencia.

Archivos de correspondencia

- Son los que indican cómo se corresponde la estructura de la base de datos con las clases del programa O-O.
- Normalmente, hay un archivo de correspondencia para cada clase y se llama ***nombreclase*`.hbm.xml`**
- Los archivos de correspondencia los veremos después. Por ahora sólo ponemos un ejemplo.

`Cat.hbm.xml`

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-
  3.0.dtd">
<hibernate-mapping>
    <class name="org.hibernate.examples.quickstart.Cat"
  table="cat">
        <id name="id" type="string" unsaved-value="null" >
            <column name="cat_id" sql-type="char(32)"
              not-null="true"/>
            <generator class="uuid.hex"/></id>
        <property name="name">
            <column name="name" length="16"
              not-null="true"/></property>
        <property name="sex"/><property name="weight"/>
    </class>
</hibernate-mapping>
```

Nota importante

- Este archivo hay que copiarlo de los archivos adjuntos a esta presentación (están en el mismo directorio que la presentación).
- No se debe copiar directamente de la transparencia pues puede tener caracteres extraños que den errores al ejecutarse.

3.3. Instalación de Hibernate

- 3.3.1. Descargar Hibernate.
- 3.3.2. Copiar JARs.
- 3.3.3. Colocar archivos de configuración.
- 3.3.4. Desplegar el ejemplo de prueba.

Se crea una BD

- Con el mismo nombre que le habíamos dado en los archivos de configuración (quickstart).
- Que tenga una tabla **InnoDB** llamada "cat" con los siguientes campos (si es un SGBD diferente de MySQL, quizás habrá que cambiar el tipo **double**).

cat_id	**varchar(32) PRIMARY KEY NOT NULL**
name	**varchar(16) NOT NULL**
sex	**char(1)**
weight	**double**

En MySQL: create table cat (cat_id varchar(32) PRIMARY KEY NOT NULL, name varchar(16) NOT NULL, sex char(1), weight double) ENGINE=InnoDB;

Creemos la clase `Cat.java` siguiente y despleguémosla en Tomcat

```
package org.hibernate.examples.quickstart;

public class Cat {
   private String id;
   private String name;
   private char sex;
   private float weight;
   public Cat() {
   }
   public String getId() {
       return id;
   }
   private void setId(String id) {
       this.id = id;
   }
```

`Cat.java` (continuación)

```
public String getName() {
   return name;
}
public void setName(String name) {
   this.name = name;
}
public char getSex() {
   return sex;
}
public void setSex(char sex) {
   this.sex = sex;
}
public float getWeight() {
   return weight;
}
public void setWeight(float weight) {
   this.weight = weight;
}}
```

Creamos una JSP con el siguiente scriptlet y la desplegamos.

```
<%@ page
   import="org.hibernate.examples.quickstart.*,
   org.hibernate.*, org.hibernate.cfg.*" %>
<%
SessionFactory sessionFactory = new
   Configuration().configure().buildSessionFactory();
Session sesion = sessionFactory.openSession();
Transaction tx= sesion.beginTransaction();
Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);
sesion.save(princess);
tx.commit();
sesion.close();
out.print("Acabado");
%>
```

(Si queremos que no dé error en Eclipse deberemos incluir hibernate3.jar con Add External JARs)

Ejecutamos la JSP

- Miramos la base de datos desde MySQL.
- Debe haberse insertado un registro con las características con el nombre de Princess y las características que hemos especificado en la JSP.

Programa del tema 3

- 3.1. Motores de persistencia.
- 3.2. Instalación de MySQL.
- 3.3. Instalación de Hibernate.
- 3.4. El patrón DAO.
- 3.5. Conceptos básicos de Hibernate.
- 3.6. El lenguaje HQL.

Mail: palasi@palasi.com Web: www.palasi.com

Patrones de diseño

- Soluciones a problemas comunes de la programación orientada a objetos.
- Cuando el programador encuentra un problema, se pregunta a que patrón de diseño pertenece y lo resuelve con la solución que tiene este patrón de diseño.
- Así los libros de patrones de diseño son como un libro de "recetas" con soluciones a problemas comunes.
- Son muy útiles a la hora de programar, aumentan la productividad, la calidad y reusabilidad del código.

El movimiento de patrones de diseño

- Comienza con "Design Patterns" de la "banda de los cuatro" ("Gang of Four"o "GoF":Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides). Presenta 23 patrones básicos ("GoF patterns"). Esta escrita para C++ pero los patrones son aplicables a cualquier lenguaje.
- A partir de este texto, el movimiento de los patrones "explota". Hay libros que definen patrones para J2EE, para EJB, antipatrones, etc.
- Incluso hay sitios web de programación que tienen secciones de patrones enviados por los usuarios.
- En este curso, sólo nos interesará un patrón: el patrón DAO (no es de la banda de los cuatro, sino un patrón J2EE).

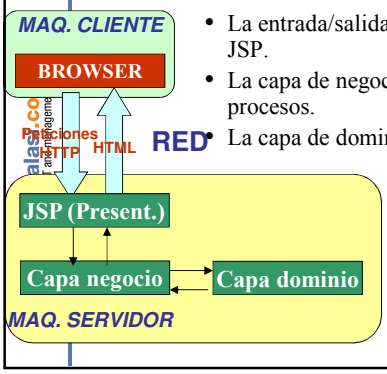
El patrón DAO

- **D**ata **A**ccess **O**bject. Sirve para aislar la implementación de persistencia del resto del programa, para dar una mayor flexibilidad.
- Así, por ejemplo, podemos hacer un programa con Hibernate y después cambiarlo sin esfuerzo a JDBC, "entity beans" o cualquier otro mecanismo de persistencia.
- Lo veremos con un ejemplo.

Ejemplo

- Una aplicación Web para ingresar dinero en una libreta desde una página Web. El usuario entra el número de libreta (en un cuadro de texto llamado "nlib")y el monto que desea depositar (en un cuadro de texto llamado "monto").
- Esta aplicación Web será parte de una aplicación bancaria más completa.

Usaremos la arquitectura que usábamos hasta ahora



- La entrada/salida se programará en la JSP.
- La capa de negocio tendrá los procesos.
- La capa de dominio tendrá los datos.

Capa de dominio: `Libreta` (simplificada)

```
package com.aurumsol.cursojava.banco.dominio;
public class Libreta {
  private int numero, saldo;

  public void iniciar(int nLibreta){
     numero = nLibreta;
     saldo = 0;
  }
  public void depositar(int monto){
     saldo += monto;
  }
}
```

- Los datos que se necesita en esta aplicación es la clase **`Libreta`**, que forma nuestra capa de dominio

Mail: palasi@palasi.com Web: www.palasi.com

Capa de negocio

- Como sabemos, la capa de negocio contiene los procesos.
- En este caso, sólo tenemos un proceso, que es el de depositar dinero en una libreta.
- Por lo tanto, lo que tendremos es una clase de negocio con un método que deposite el dinero en una libreta.

Es decir, habría algo como lo siguiente

```
package com.aurumsol.cursojava.banco.negocio;
import com.aurumsol.cursojava.banco.dominio.*;
public class NgcBanco {
  public boolean ingresar(int numLib, int monto){

  //Aquí el código que deposita en la libreta.


  }
}
```

Normalmente, habrá más métodos en esta clase, pero por ahora nos centraremos en este.

Paréntesis: como usaremos la clase `Libreta`, que está en otro paquete

```
package com.aurumsol.cursojava.banco.negocio;
import com.aurumsol.cursojava.banco.dominio.*;
public class NgcBanco {
  public boolean ingresar(int numLib, int monto){

  //Aquí el código que deposita en la libreta.


  }
}
```

Debemos poner el import.

Recordemos: Importar es

- especificar que usaremos en una clase, otra clases que no pertenecen al mismo paquete.
- En las JSP usábamos

```
<%@ page
  import="nombrepaquete.NombreClase">
<%@ page import="nombrepaquete.*">
```

- En las clases se pone:

```
import nombrepaquete.NombreClase;
import nombrepaquete.*;
```

- Los `import` están entre `package` y `public class`.

El código del método `ingresar` puede ser así

```
package com.aurumsol.cursojava.banco.negocio;
import com.aurumsol.cursojava.banco.dominio.*;
public class NgcBanco {
  public boolean ingresar(int numLib, int monto){
     Libreta libreta1 = Obtiene libreta1 de BD
     if (se encuentra){
          libreta1.depositar(monto);
          Actualiza la libreta1 en la BD
          return true;
     }else{
          return false;
     }
  }
}
```

Por ahora dejamos las partes en rojo, que veremos después.

La capa de presentación

- Constará de
  - un documento HTML con el formulario para recoger los datos (número de libreta y monto a depositar).
  - una JSP que llama a las capas inferiores para realizar el depósito e informa al usuario.
- El formulario no lo especificamos aquí: es obvio. Sólo hace falta saber que los cuadros de texto se llamarán "nlib" y "monto".

La JSP sería la siguiente:

```
<%@ page
import="com.aurumsol.cursojava.banco.negocio.*" %>
<%int numLib = Integer.parseInt(
request.getParameter("nlib"));
int monto = Integer.parseInt(
request.getParameter("monto"));
NgcBanco ngcBanco = new NgcBanco();
boolean exito = ngcBanco.ingresar(numLib,monto);%>
<HTML><HEAD><TITLE>Ingresar</TITLE></HEAD><BODY>
<%if (exito) {
     out.print("Depósito efectuado");
}else {
     out.print("Libreta no encontrada");
}%></BODY></HTML>
```

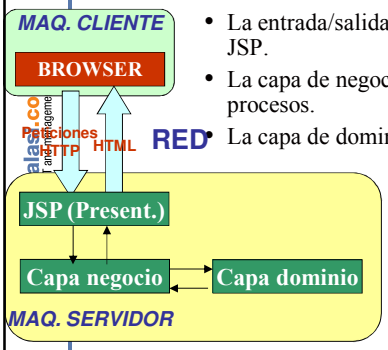
Como vemos, esta JSP sólo hace tareas de entrada/salida

```
<%@ page
import="com.aurumsol.cursojava.banco.negocio.*" %>
<%int numLib = Integer.parseInt(
request.getParameter("nlib"));
int monto = Integer.parseInt(
request.getParameter("monto"));
NgcBanco ngcBanco = new NgcBanco();
boolean exito = ngcBanco.ingresar(numLib,monto);%>
<HTML><HEAD><TITLE>Ingresar</TITLE></HEAD><BODY>
<%if (exito) {
     out.print("Depósito efectuado");
}else {
     out.print("Libreta no encontrada");
}%></BODY></HTML>
```

Recoge la entrada, llama a la capa de negocio para el cálculo y emite la salida

```
<%@ page
import="com.aurumsol.cursojava.banco.negocio.*" %>
<%int numLib = Integer.parseInt(
request.getParameter("nlib"));
int monto = Integer.parseInt(
request.getParameter("monto"));
NgcBanco ngcBanco = new NgcBanco();
boolean exito = ngcBanco.ingresar(numLib,monto);%>
<HTML><HEAD><TITLE>Ingresar</TITLE></HEAD><BODY>
<%if (exito) {
     out.print("Depósito efectuado");
}else {
     out.print("Libreta no encontrada");
}%></BODY></HTML>
```

Como vemos, hemos usado la arquitectura en n-capas



- La entrada/salida se programará en la JSP.
- La capa de negocio tendrá los procesos.
- La capa de dominio tendrá los datos.

Sólo habíamos dejado pendiente las partes en rojo

```
package com.aurumsol.cursojava.banco.negocio;
import com.aurumsol.cursojava.banco.dominio.*;
public class NgcBanco {
  public boolean ingresar(int numLib, int monto){
     Libreta libreta1 = Obtiene libreta1 de BD
     if (se encuentra){
          libreta1.depositar(monto);
          Actualiza la libreta1 en la BD
          return true;
     }else{
          return false;
     }
  }
}
```

Observemos

- Fijémonos que lo que se guarda en la BD son libretas. Son objetos. Esto es porque tenemos un motor de persistencia (podemos tratar la BD como si fuera BD O-O).

Mail: palasi@palasi.com Web: www.palasi.com

Hay dos operaciones de la base de datos sobre libretas

```
package com.aurumsol.cursojava.banco.negocio;
import com.aurumsol.cursojava.banco.dominio.*;
public class NgcBanco {
  public boolean ingresar(int numLib, int monto){
     Libreta libreta1 = Obtiene libreta1 de BD
     if (se encuentra){
          libreta1.depositar(monto);
          Actualiza la libreta1 en la BD
          return true;
     }else{
          return false;
     }
  }
}
```

Serán dos métodos: para actualizar y recuperar libretas de la BD

```
package com.aurumsol.cursojava.banco.negocio;
import com.aurumsol.cursojava.banco.dominio.*;
public class NgcBanco {
  public boolean ingresar(int numLib, int monto){
     Libreta libreta1 = Obtiene libreta1 de BD
     if (se encuentra){
          libreta1.depositar(monto);
          Actualiza la libreta1 en la BD
          return true;
     }else{
          return false;
     }
  }
}
```

El patrón DAO dice

- Que todos los métodos para acceder a objetos de la misma clase de dominio en la base de datos, deben agruparse en una misma clase de datos (llamada clase DAO o DAO).
- En nuestro caso, tendremos que todos los métodos para actualizar y recuperar libretas en la base de datos deben ir en una única clase, que llamaremos DAOLibreta.

En nuestro caso, tendríamos

```
package com.aurumsol.cursojava.banco.datos;
import com.aurumsol.cursojava.banco.dominio.*;
public class DAOLibreta {
  public void actualizar(Libreta libreta){
     //Aquí un código
  }
  public Libreta obtener(int numLib){
     //Aquí un código
  }
}
```

Nota: si obtener no encuentra la libreta devolverá `null`

La clase de negocio quedaría

```
package com.aurumsol.cursojava.banco.negocio;
import com.aurumsol.cursojava.banco.datos.*;
import com.aurumsol.cursojava.banco.dominio.*;
public class NgcBanco {
  public boolean ingresar(int numLib, int monto){
     DAOLibreta daoLib = new DAOLibreta();
     Libreta libreta1=daoLib.obtener(numLib);
     if (libreta1!=null){
          libreta1.depositar(monto);
          daoLib.actualizar(libreta1);
          return true;
     }else{
          return false;
     }
  }
}
```

Comprendan este método

```
package com.aurumsol.cursojava.banco.negocio;
import com.aurumsol.cursojava.banco.datos.*;
import com.aurumsol.cursojava.banco.dominio.*;
public class NgcBanco {
  public boolean ingresar(int numLib, int monto){
     DAOLibreta daoLib = new DAOLibreta();
     Libreta libreta1=daoLib.obtener(numLib);
     if (libreta1!=null){
          libreta1.depositar(monto);
          daoLib.actualizar(libreta1);
          return true;
     }else{
          return false;
     }
  }
}
```

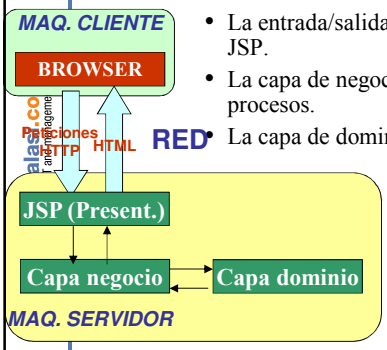
Hemos delegado toda la tarea de acceder la base de datos al DAO

```
package com.aurumsol.cursojava.banco.negocio;
import com.aurumsol.cursojava.banco.datos.*;
import com.aurumsol.cursojava.banco.dominio.*;
public class NgcBanco {
  public boolean ingresar(int numLib, int monto){
      DAOLibreta daoLib = new DAOLibreta();
      Libreta libreta1=daoLib.obtener(numLib);
      if (libreta1!=null){
            libreta1.depositar(monto);
            daoLib.actualizar(libreta1);
            return true;
      }else{
            return false;
      }
  }
}
```

¿En qué capa se encuentra `DAOLibreta`?

- Buena pregunta.

En la arquitectura en n-capas



- La entrada/salida se programará en la JSP.
- La capa de negocio tendrá los procesos.
- La capa de dominio tendrá los datos.

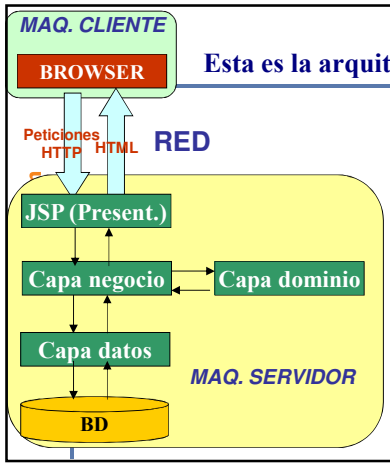
La clase `DAOLibreta`

- No pertenece a la capa de presentación pues no trata de la entrada/salida.
- No pertenece a la capa de dominio pues no es un dato de la aplicación.
- No pertenece a la capa de negocio pues no efectúa ningún proceso de datos.
- Lo único que hace es acceder a la base de datos.

Hay otra capa aparte de las tres que hemos visto

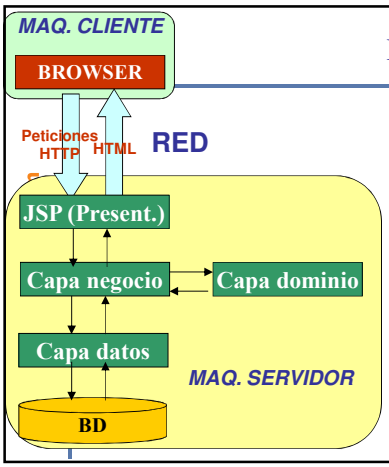
- Es la capa de datos y se dedica a acceder a la base de datos (guardar y recuperar objetos de la base).
- Es esa capa la que tendrá todas las clases que implementan el patrón DAO.

Ahora sí. Esta es la arquitectura en n-capas



- Son cuatro capas: presentación, negocio, dominio y datos.
- La base de datos puede estar en la misma máquina que las capas del programa o en otra.

En nuestro caso

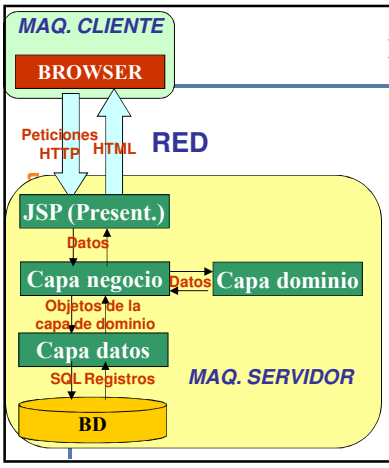


- La capa de datos está compuesta por las clases DAO.
- Son ellas las únicas que acceden a la base de datos.

Una ventaja de esto

- Los DAO son independientes del mecanismo de persistencia que utilicemos.
- Si queremos usar un mecanismo de persistencia diferente de Hibernate (otros motores de persistencia o JDBC), sólo debemos cambiar la capa de datos (los DAO).

Datos entre capas



- La capa de datos envía SQL a la base y esta le devuelve registros.
- La capa de datos convierte estos registros en objetos de la capa de dominio.
- Entre la capa de negocio y la de datos se intercambian objetos de la capa de dominio.

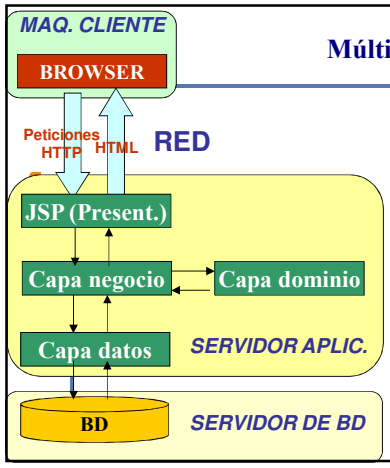
Importante

- Los objetos de la capa de dominio son los que se guardan en la base de datos.
- Así, en nuestro caso, los objetos de clase `Libreta` será lo que se guardará en la BD.
- A estos objetos se les llama a veces "objetos persistentes" y a sus clases se les llama "clases persistentes".

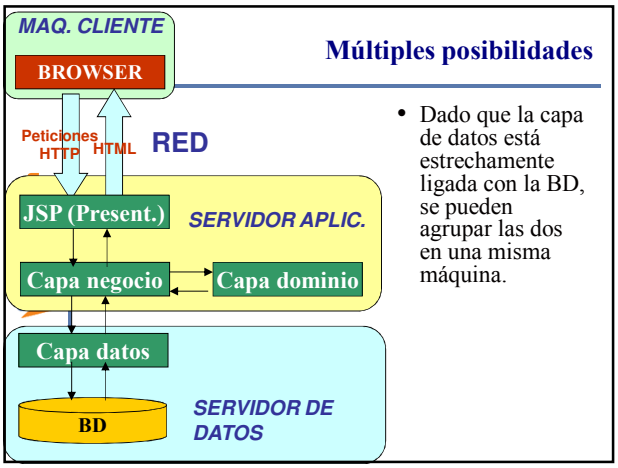
Algunas ventajas de la arquitectura en n-capas

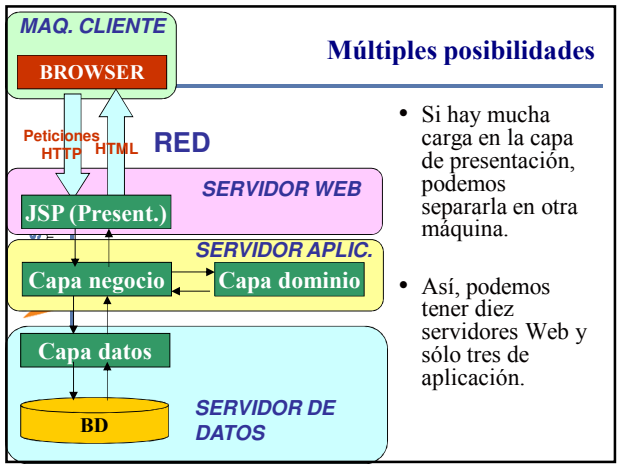
- El código es más limpio:
  - El acceso a la base de datos está en un sitio (DAO).
  - Los procesos en otro sitio (las clases de negocio).
  - Los datos en otro (la capa de dominio).
  - La presentación en otro (las JSP).
- El código es más legible, más fácil de depurar y mantener.
- Por ejemplo, si creemos que el problema está en la BD, podemos hacer que el DAO devuelva objetos de prueba para aislar el problema.
- Además, esta arquitectura nos da más posibilidades de distribuir nuestra aplicación y balancear su carga.

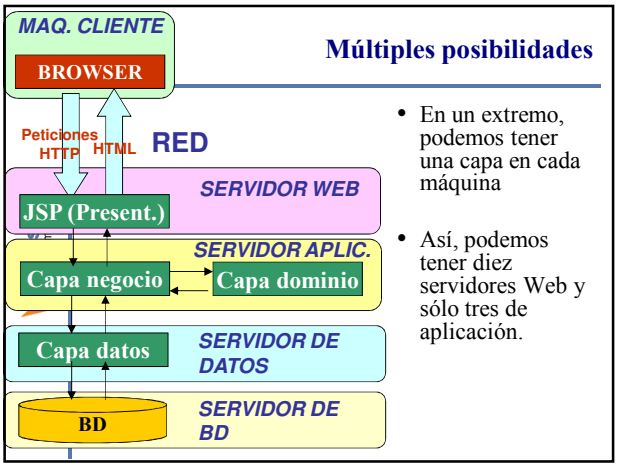
Múltiples posibilidades

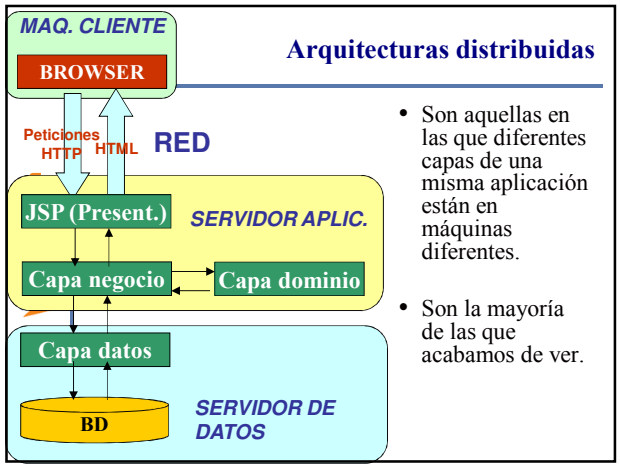


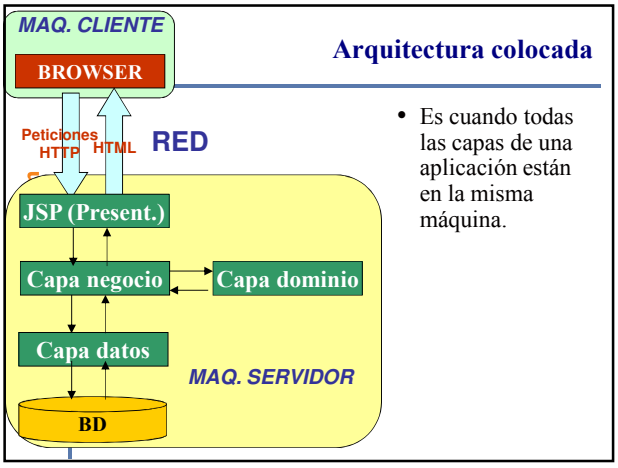
- Podemos poner la base de datos en otra máquina.
- Esto es útil sobre todo cuando tenemos varios servidores de aplicaciones que acceden a la misma base de datos.

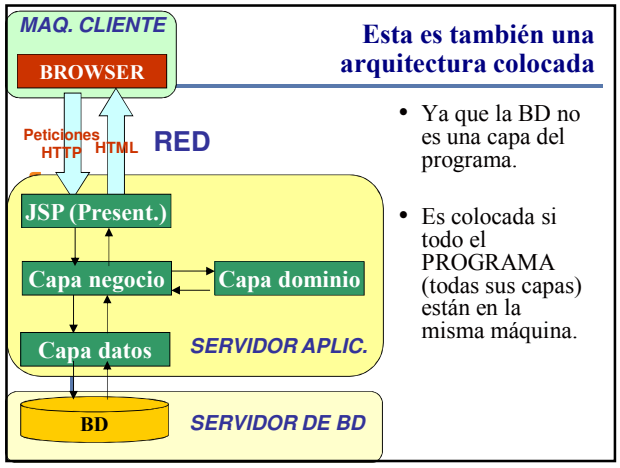












Mail: palasi@palasi.com Web: www.palasi.com

Arquitectura distribuida o colocada

- Ventajas de la arquitectura distribuida:
  - A cada capa le podemos asignar las máquinas que necesita.
  - Entre las diferentes máquinas podemos instalar firewalls adicionales con lo que se aumenta la seguridad.
  - Hay veces en que diferencias entre vendedores fuerzan a una aplicación distribuida.
- Desventajas de la arquitectura distribuida:
  - Programación mucho más compleja.
  - Más carga al intentar coordinar la información entre máquinas.

Arquitectura distribuida o colocada

- "La primera ley del diseño distribuido de objetos es: ¡No distribuyas tus objetos! Vende a tu abuela favorita primero, si puedes." – Martin Fowler, Patterns of Enterprise Application Architecture.
- "El principio primario en la distribución de objetos es la parsimonia: distribuye tan poco como puedas" – Colleen Roe, Sergio Gonik, Server-Side Design Principles for Scalable Internet Systems.
- En el 95% de los casos, es mejor la arquitectura colocada.
- En el otro 5%, Java proporciona herramientas para una arquitectura distribuida.

Con esto

- Ya pueden ver cómo se estructura una aplicación real con n-capas.
- Ahora sólo nos queda implementar la capa de datos.
- Esto lo haremos con Hibernate.

Programa del tema 3

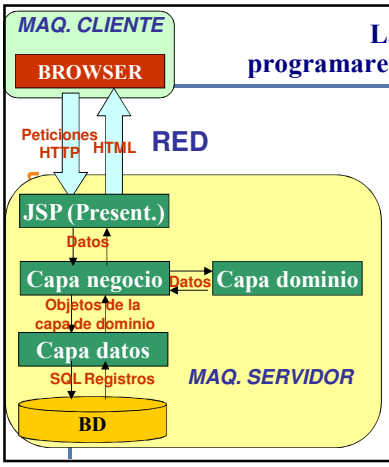
- 3.1. Motores de persistencia.
- 3.2. Instalación de MySQL.
- 3.3. Instalación de Hibernate.
- 3.4. El patrón DAO.
- 3.5. Conceptos básicos de Hibernate.
- 3.6. El lenguaje HQL.

Ya casi habíamos visto todo el código de nuestra aplicación

```
package com.aurumsol.cursojava.banco.datos;
import com.aurumsol.cursojava.banco.dominio.*;
public class DAOLibreta {
  public void actualizar(Libreta libreta){
     //Aquí un código
  }
  public Libreta obtener(int numLib){
     //Aquí un código
  }
}
```

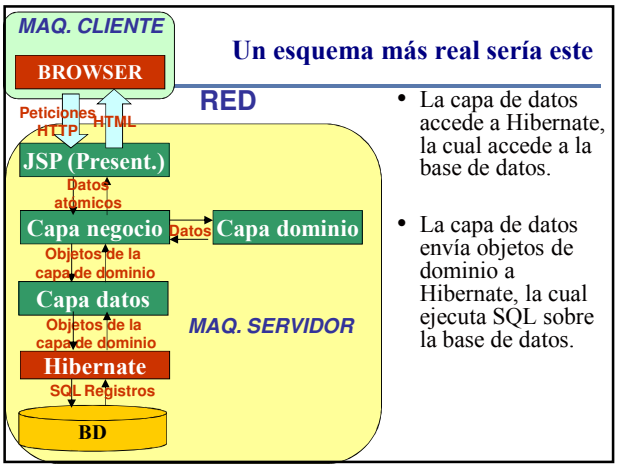
- Sólo nos faltaba programar la capa de datos (el DAO). Lo haremos con Hibernate.

La capa de datos la programaremos en Hibernate



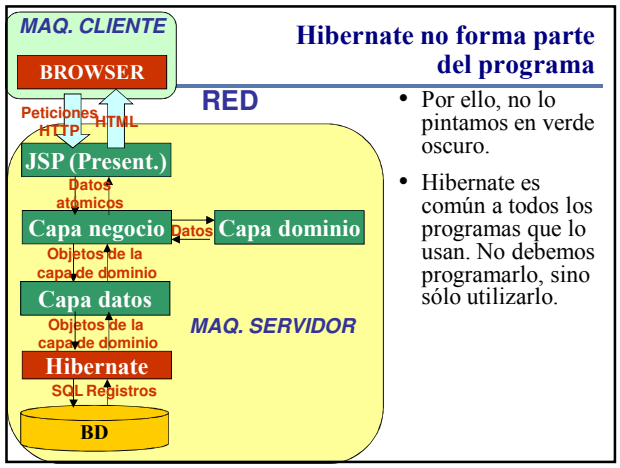
- Esto quiere decir que la capa de datos no accede directamente a la base de datos como aparece en el esquema.
- En realidad, este esquema está simplificado.
- La capa de datos llama a Hibernate, no accede directamente a la base.

Un esquema más real sería este



- La capa de datos accede a Hibernate, la cual accede a la base de datos.
- La capa de datos envía objetos de dominio a Hibernate, la cual ejecuta SQL sobre la base de datos.

Hibernate no forma parte del programa



- Por ello, no lo pintamos en verde oscuro.
- Hibernate es común a todos los programas que lo usan. No debemos programarlo, sino sólo utilizarlo.

Programar en Hibernate

- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

Programar en Hibernate

- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

Adaptar la clase persistente a Hibernate

- La clase persistente es la clase de la capa de dominio que vamos a guardar en la BD.
- Son las de la capa de dominio.
- En nuestro caso, es la clase **`Libreta`**.
- Vamos a adaptarla para que pueda trabajar con Hibernate.

¿Es necesario adaptar la clase persistente para Hibernate?

- En realidad, no.
- Cualquier clase que tengamos se puede tratar con Hibernate sin hacer ningún cambio.
- Los cambios que veremos aquí son **recomendados, no obligatorios**.

Una adaptación conveniente

- La adaptación que vamos a explicar tampoco es obligatoria para lo que haremos aunque sí para algunas funciones más avanzadas de Hibernate.
- Es una adaptación ampliamente seguida por los programadores que usan Hibernate.

La adaptación se trata

- De crear un atributo para almacenar la clave primaria de la tabla. Lo llamaremos "atributo identidad". Normalmente, se llamará "**`id`**".
- Este atributo deberá ser privado.

`Libreta` queda así

```
package com.aurumsol.cursojava.banco.dominio;
public class Libreta {
  private int id; // Campo identidad
  private int numero, saldo;
  public void iniciar(int nLibreta){
     numero = nLibreta;
     saldo = 0;
  }
  public void depositar(int monto){
     saldo += monto;
  }
}
```

Programar en Hibernate

- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

Se necesitan tres tipos de archivos de configuración

- El archivo **`context.xml`** que indica la conexión de la base de datos.
- El archivo donde se especifican la configuración de Hibernate.
  - Sólo hay uno.
  - Su nombre es **`hibernate.cfg.xml`**
- Los archivos de correspondencia que indican las clases que se van a persistir
  - Hay uno por cada clase persistente.
  - Su nombre es ***NombreClase*`.hbm.xml`**

Se necesitan tres tipos de archivos de configuración

- El archivo **`context.xml`** que indica la conexión de la base de datos.
- El archivo donde se especifican la configuración de Hibernate.
  - Sólo hay uno.
  - Su nombre es **`hibernate.cfg.xml`**
- Los archivos de correspondencia que indican las clases que se van a persistir
  - Hay uno por cada clase persistente.
  - Su nombre es ***NombreClase*`.hbm.xml`**

Creamos `context.xml`

```
<Context path="" docBase="ROOT">
<Resource name="jdbc/nombreJNDI"
  scope="Shareable"
  type="javax.sql.DataSource"
  factory="org.apache.tomcat.dbcp.dbcp.
  BasicDataSourceFactory"
  url="URLdelaBD"
  driverClassName="ClaseDelDriver"
  username="usernameBD"
  password="passwordBD" maxWait="3000"
  maxIdle="100" maxActive="10">
</Resource>
</Context>
```

- El archivo se colocará en **`webapps\ROOT\META-INF`** del directorio de instalación de Tomcat.

En el anterior archivo

- *nombreJNDI:* Es un nombre arbitrario pero es bueno que coincida con el nombre de la BD.
- *URLdelaBD*: URL para acceder a la BD. En MySQL es **`jdbc:mysql://localhost/`*`nombre`***, donde ***nombre*** es el nombre de la BD.
- *ClaseDelDriver*: es la clase del driver JDBC. En MySQL es **`com.mysql.jdbc.Driver`**.
- *usernameBD* y *passwordBD*: son el username y el password para acceder a la BD.

Por ejemplo, para la base de datos "prueba" puede ser así

```
<Context path="" docBase="ROOT">
<Resource name="jdbc/prueba"
  scope="Shareable"
  type="javax.sql.DataSource"
  factory="org.apache.tomcat.dbcp.dbcp.BasicD
  ataSourceFactory"
  url="jdbc:mysql://localhost/prueba"
  driverClassName="com.mysql.jdbc.Driver"
  username="root" password="admin"
  maxWait="3000" maxIdle="100"
  maxActive="10">
</Resource>
</Context>
```

- El archivo se colocará en **`webapps\ROOT\META-INF`** del directorio de instalación de Tomcat.

Se necesitan tres tipos de archivos de configuración

- El archivo **`context.xml`** que indica la conexión de la base de datos.
- El archivo donde se especifican la configuración de Hibernate.
  - Sólo hay uno.
  - Su nombre es **`hibernate.cfg.xml`**
- Los archivos de correspondencia que indican las clases que se van a persistir
  - Hay uno por cada clase persistente.
  - Su nombre es ***`NombreClase`*`.hbm.xml`**

`hibernate.cfg.xml` (colocar en el directorio de clases)

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-
  configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.datasource">
         java:comp/env/jdbc/nombreJNDI </property>
        <property name="show_sql">false</property>
        <property name="dialect"> dialecto</property>
           archivos de correspondencia de cada clase persist.
    </session-factory>
</hibernate-configuration>
```

En el archivo anterior

- *nombreJNDI*: Es el nombre que hemos puesto en el archivo de configuración XML que hay en **`conf\Catalina\localhost`** (mirar antes).
- *dialecto*: Dialecto del SQL de la BD. Se encuentra en la documentación de Hibernate. Para MySQL es **`org.hibernate.dialect.MySQLDialect`**
- *archivos de correspondencia* : son un conjunto de líneas

`<mapping resource="`*`nombre1`*`"/>`

...

`<mapping resource="`*`nombreN`*`"/>`

donde ***nombre1***,..., ***nombreN*** son los nombres de los archivos de correspondencia. Hay un archivo por cada clase persistente.

Mail: palasi@palasi.com Web: www.palasi.com

Por ejemplo, para nuestro caso

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-
  configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.datasource">
         java:comp/env/jdbc/prueba</property>
        <property name="show_sql">false</property>
        <property name="dialect">
        org.hibernate.dialect.MySQLDialect</property>
     <mapping resource="Libreta.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Se necesitan tres tipos de archivos de configuración

- El archivo `context.xml` que indica la conexión de la base de datos.
- El archivo donde se especifican la configuración de Hibernate.
  - Sólo hay uno.
  - Su nombre es `hibernate.cfg.xml`
- Los archivos de correspondencia que indican las clases que se van a persistir
  - Hay uno por cada clase persistente.
  - Su nombre es *NombreClase*`.hbm.xml`

Archivo de correspondencia para una clase persistente

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD
  3.0//EN"
  "http://hibernate.sourceforge.net/hibern
  ate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="clase" table="tabla">
     atributos
  </class>
</hibernate-mapping>
```

En el archivo anterior

- ***clase***: Clase que queremos persistir con el formato ***nombrepaquete.nombreClase***
- ***tabla:*** Nombre de la tabla de la base de datos (podemos poner cualquier nombre pero después hemos de usar ese nombre en la base de datos).
- ***atributos***: especificación de atributos de la clase.

Para cada atributo

- Si es un atributo diferente del "atributo identidad":

```
<property name="nombreAtributo"
  access="field"/>
```

- Donde ***nombreAtributo*** es el nombre del atributo que coincidirá con el nombre del campo de la tabla de la BD donde se guarda.
- También puede ser que sean nombres diferentes, pero la sintaxis en este caso no la veremos por ahora.

Si es el atributo identidad

```
<id name="nombreAtributo" type="int"
  access="field">
  <column name="nombreCampo"/>
  <generator class="identity"/>
</id>
```

- ***nombreAtributo*** es el nombre del atributo identidad.
- ***nombreCampo*** es el nombre del campo en que se guarda este atributo. Es decir, la clave primaria.

En nuestro caso, `Libreta.hbm.xml`

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-
  mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.aurumsol.cursojava.
   banco.dominio.Libreta" table="libreta">
    <id name="id" type="int" access="field">
       <column name="id"/>
       <generator class="identity"/>
    </id>
    <property name="numero" access="field"/>
    <property name="saldo" access="field"/>
  </class>
</hibernate-mapping>
```

¿Cuándo se cambian estos archivos?

- El archivo que indica la conexión de la base de datos (**`context.xml`**).

Si cambia la BD, el contexto, clave o contraseña

- El archivo donde se especifican la configuración de Hibernate (**`hibernate.cfg.xml`**).

Si cambia la BD o el conjunto de clases persistentes

- Los archivos de correspondencia que indican las clases que se van a persistir

Cada vez que cambian las clases persistentes

Pero la base de datos cambia con poca frecuencia

- Normalmente lo que cambia es:
  - Los archivos de correspondencia.
  - La lista de clases persistentes en **`hibernate.cfg.xml`**.
- Aún esto, puede generarse automáticamente con herramientas como Xdoclet o Middlegen o plugins de Eclipse para Hibernate.

¿Dónde se despliegan estos archivos?

- El archivo que indica la conexión de la base de datos (**`context.xml`**).

`webapps\ROOT\META-INF`

- El archivo donde se especifican la configuración de Hibernate (**`hibernate.cfg.xml`**)

`webapps\ROOT\WEB-INF\classes`

- Los archivos de correspondencia que indican las clases que se van a persistir

`webapps\ROOT\WEB-INF\classes`

Programar en Hibernate

1. Adaptar la clase persistente a Hibernate.
2. Escribir los archivos de configuración.
3. Crear el esquema de la BD.
4. Programar el acceso a la BD.
5. Desplegar.

Primero, creamos la base de datos que contendrá las clases que vamos a guardar

- Hibernate tiene herramientas para hacerlo automáticamente a partir de los archivos de correspondencia (e incluso para crear las clases persistentes a partir de los archivos de correspondencia).
- Pero nosotros lo haremos a mano.

Creemos una base de datos "prueba"

- Creemos una tabla llamada "libreta".
- Allí guardaremos nuestras libretas.

¿Qué campos debe tener una tabla que guarda una clase?

- Debe tener un campo para guardar cada atributo de la clase.
- El campo para el atributo "identidad" debe ser de tipo "autoincremento" (llamado en algunos SGBDs, "identity").

En nuestro caso, ¿qué campos debe tener la tabla que guarda `Libreta`?

- Si la clase **`Libreta`** comienza así:

```
package
  com.aurumsol.cursojava.banco.dominio;
public class Libreta {
  private int id; // Campo identidad
  private int numero, saldo;
```

- Deberemos tener dos campos enteros: **`numero`** y **`saldo`**.
- Además, tendremos un campo autoincremento que será clave primaria. Lo llamaremos "**`id`**".

En resumen, en la tabla "libreta" tenemos

`id`	`integer NOT NULL AUTO_INCREMENT`
`numero`	`integer NOT NULL`
`saldo`	`integer NOT NULL`

- Desde MySQL Command Line Client

```
CREATE TABLE libreta (
  id INTEGER NOT NULL AUTO_INCREMENT,
  numero INTEGER NOT NULL, saldo INTEGER
  NOT NULL,  PRIMARY KEY(`id`)) ENGINE =
  InnoDB;
```

Nota: En MySQL es bueno que definir las tablas como InnoBD

Creando la tabla.

- Desde MySQL Query Browser

Column Name	Datatype	NOT NULL	AUTO INC	Flags	Default Value
id	INTEGER	✓	✓	UNSIGNED ZEROFILL	NULL
numero	INTEGER	✓		UNSIGNED ZEROFILL	NULL
saldo	INTEGER	✓		UNSIGNED ZEROFILL	NULL

- Otras formas:
  - **`SchemaExport`** y **`SchemaUpdate`** crean automáticamente las instrucciones como CREATE TABLE a partir de los archivos de configuración.
  - Hay plugins de Eclipse que van más alla y crean automáticamente la tabla a partir de los archivos de configuración.

Nota

- Fíjense que el nombre de los campos de la base de datos es exactamente el mismo que el nombre de los atributos de la clase: **`id`**, **`numero`**, **`saldo`**.
- En realidad, no tiene porque ser así, pero es lo más sencillo y la opción por defecto.
- Es la opción que usaremos aquí.

Programar en Hibernate

- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

Conceptos básicos de Hibernate

- **`Session`** (Sesión). Una conexión de corta duración a la base de datos. Normalmente, cuando queremos hacer una operación en la base de datos.
  - Obtenemos una sesión.
  - Hacemos la operación.
  - Cerramos la sesión lo más pronto posible.
- **`SessionFactory`** (fábrica de sesiones). Una clase que nos permite obtener sesiones.

Esquema de programación en Hibernate

- 1. Obtener una **`SessionFactory`**.
- 2. Obtener una **`Session`**.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

Esquema de programación en Hibernate

- 1. Obtener una **`SessionFactory`**.
- 2. Obtener una **`Session`**.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

Obtener una `SessionFactory`

- Hay varias formas. La que usaremos es:

```
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
```

- Así se creará una SessionFactory con las opciones de configuración que hay en el archivo **`hibernate.cfg.xml`** (lo veremos después)
- Nota: Para usar estos objetos debemos:

```
import org.hibernate.*
import org.hibernate.cfg.*
```

Sólo se debería obtener una única `SessionFactory` para todo el programa

```
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
```

- Sin embargo, por ahora no sabemos cómo hacer esto. Requiere de un patrón llamado "Singleton" que no hemos visto.
- Por eso, obtendremos una cada vez que accedamos a la BD.
- Esto es más lento, pero, por ahora servirá.

Esquema de programación en Hibernate

- 1. Obtener una `SessionFactory`.
- 2. Obtener una `Session`.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

Obteniendo una `Session`

```
Session sesion =
  sessionFactory.openSession();
```

- Hay que obtener una sesión cada vez que accedemos a la BD.
- Una sesión debería durar poco.
- Normalmente, usaremos una sesión para cada método de la clase DAO (más adelante, se verán mejores formas).
  - La abriremos al principio del método.
  - La cerraremos al final del método.

Esquema de programación en Hibernate

- 1. Obtener una `SessionFactory`.
- 2. Obtener una `Session`.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

Operaciones sobre la base de datos

- Guardar un objeto nuevo

```
sesion.save(objeto)
```

- Actualizar un objeto que ya existía.

```
sesion.update(objeto)
```

- Borrar un objeto.

```
sesion.delete(objeto)
```

Operaciones sobre la base de datos

- Recuperar objetos de la base de datos.

```
List lista =
  sesion.createQuery(condicion).list();
```

- Recupera todos los objetos de la base de datos que cumplen la condición y los guarda en una lista. Podría hacerse en dos pasos.

```
Query consulta =
  sesion.createQuery(condicion);
List lista = consulta.list()
```

- `Query` es una clase para consultas a la base de datos.

Las listas

- Objetos del tipo `List`. Métodos:
- Si *l* es de tipo lista
- `l.isEmpty()` Dice si la lista está vacía.
- `(Clase) l.get(p)` Obtiene el objeto que está en la posición *p* (comienzan de 0) y con la clase *Clase*
- `l.size()` Obtiene el número de elementos de la lista.
- La clase `List` pertenece al paquete `java.util`, por eso se debe importar.

Mail: palasi@palasi.com Web: www.palasi.com

Las condiciones

```
List lista =
  sesion.createQuery(condicion).list();
```

- La condición es un **`String`** que expresa qué objetos queremos recuperar de la base de datos.
- Está en un lenguaje parecido al SQL llamado HQL (**H**ibernate **Q**uery **L**anguage)

Una condición en HQL

```
from Clase as objeto where condiciones
```

Ejemplos

```
from Libreta as libreta where
  libreta.saldo>1000
```

```
from Libreta as libreta where
  libreta.numero=123
```

Por ejemplo

```
List lista1 =
  sesion.createQuery("from Libreta as
  libreta where
  libreta.saldo>1000").list();
```

- En la **`lista1`** se almacenará una lista con todas las libretas con saldo superior a 1000.

Esquema de programación en Hibernate

- 1. Obtener una **`SessionFactory`**.
- 2. Obtener una **`Session`**.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

Guardar los cambios de la sesión (a veces)

- Si hemos grabado, actualizado o borrado.
- No si sólo hemos consultado.
- Fácil, se hace:

```
sesion.flush();
```

Esquema de programación en Hibernate

- 1. Obtener una **`SessionFactory`**.
- 2. Obtener una **`Session`**.
- 3. Hacer las operaciones.
- 4. Guardar los cambios de la sesión (a veces).
- 5. Cerrar sesión.

Cerrar sesión

- Esto se consigue:

```
sesion.close();
```

Ejemplo: guardar un nuevo objeto (creado con *objeto* = new *Clase()*)

```
//1. Obtener SessionFactory
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
//2. Obtener Session
Session sesion = sessionFactory.openSession();
//3. Hacer operaciones
sesion.save(objeto);
//4. Guardar cambios
sesion.flush();
//5. Cerrar la sesión
sesion.close();
```

Ejemplo: recuperar las cuentas con saldo mayor de 1000

```
//1. Obtener SessionFactory
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
//2. Obtener Session
Session sesion = sessionFactory.openSession();
//3. Hacer operaciones
List cuentasGordas = sesion.createQuery("from
  Libreta as libreta where
  libreta.saldo>1000").list();
//4. No se guardan cambios porque no los hay
//5. Cerrar la sesión
sesion.close();
```

Programemos el DAO

```
package com.aurumsol.cursojava.banco.datos;
import com.aurumsol.cursojava.banco.dominio.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;
public class DAOLibreta {
  public void actualizar(Libreta libreta){
     //Aquí un código
  }
  public Libreta obtener(int numLib){
     //Aquí un código
  }
}
```

- Por motivos de espacio pondremos un método por cada transparencia.

El método que actualiza la libreta

```
public void actualizar(Libreta libreta){
  SessionFactory sessionFactory = new
   Configuration().configure().
   buildSessionFactory();
  Session sesion = sessionFactory.openSession();
  sesion.update(libreta);
  sesion.flush();
  sesion.close();
}
```

El método que obtiene la libreta

```
public Libreta obtener(int numLib){
  SessionFactory sessionFactory = new
   Configuration().configure().
   buildSessionFactory();
  Session sesion = sessionFactory.openSession();
  List libretas = sesion.createQuery("from
  Libreta as libreta where
  libreta.numero="+numLib).list();
  sesion.close();
  if (libretas.isEmpty()){
     return null;
  }else{
     return (Libreta)libretas.get(0);
  }
}
```

Tenemos que importar las clases de los otros paquetes

```
package com.aurumsol.cursojava.banco.datos;
import com.aurumsol.cursojava.banco.dominio.*;
import java.util.List;
import org.hibernate.*;
import org.hibernate.cfg.*;
public class DAOLibreta {
  public void actualizar(Libreta libreta){
     //Aquí el código que hemos visto
  }
  public Libreta obtener(int numLib){
     //Aquí el código que hemos visto
  }
}
```

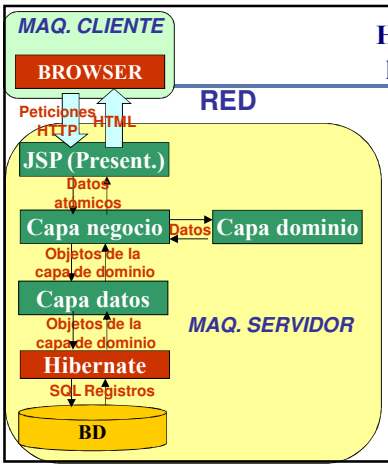
Programar en Hibernate

- 1. Adaptar la clase persistente a Hibernate.
- 2. Escribir los archivos de configuración.
- 3. Crear el esquema de la BD.
- 4. Programar el acceso a la BD.
- 5. Desplegar.

Desplegar

- Desplegamos la JSP en **webapps\ROOT**
- Desplegamos todas las clases como archivos JAR en **webapps\ROOT\WEB-INF\lib**.
- Colocamos todos los archivos de configuración donde se han dicho.
- Nos aseguramos que el servidor de la BD está funcionando.
- Ejecutamos.

Hemos acabado de programarlo todo



- La capa de datos accede a Hibernate.
- Sin embargo, Hibernate no forma parte del programa ni debemos programarlo.
- Por lo tanto, no lo pintamos en verde oscuro, pues no se considera una capa del programa.
- Hemos acabado.

Paréntesis: Plugins de Eclipse para Hibernate

- Aquí hemos escrito las clases persistentes, la base de datos y los archivos de configuración a mano.
- Esto es sencillo, pero no tiene por qué ser así.
- Se pueden generar automáticamente los archivos de configuración con herramientas como Ant, XDoclet.
- También Hibernate incluye herramientas de línea de comandos como SchemaExport y SchemaUpdate que, a partir de los archivos de configuración, construyen el DDL de la base de datos.
- También se pueden usar plugins de Eclipse para Hibernate, que no veremos aquí por falta de tiempo.

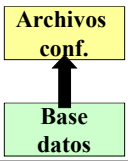
Paréntesis: Plugins de Eclipse para Hibernate

- Hibernate Synchronizer. http://www.binamics.com/hibernatesync/
  - Crea y actualiza las clases Java a partir de los archivos de configuración.
  - Adecuado si se sigue un enfoque basado en los archivos.

Mail: palasi@palasi.com Web: www.palasi.com

Paréntesis: Plugins de Eclipse para Hibernate

- Hiberclipse http://hiberclipse.sourceforge.net/
  - Genera y actualiza los archivos de configuración a partir de la base de datos.
  - Es más adecuado con un enfoque centrado en la BD, sobre todo, cuando queremos aplicar Hibernate a una base de datos preexistente.



Paréntesis: Plugins de Eclipse para Hibernate

- Hibernator. http://hibernator.sourceforge.net/
  - Va en la dirección opuesta, creando y actualizando los archivos de configuración a partir de las clases persistentes y, a partir de allí, generando la base de datos.
  - Es más adecuado con un enfoque centrado en el código Java, que es el que hemos utilizado aquí.
  - Descontinuado.

Paréntesis: Plugins de Eclipse para Hibernate

- Hibernate Tools. http://www.hibernate.org/255.html
- Crea y actualiza archivos de configuración a partir de las clases Java.
- Crea y actualiza archivos de configuración y clases Java a partir de la base de datos.
- Permite múltiples enfoques.
- En versión alfa.

Ejercicio

- Queremos programar una aplicación Web de parqueo. La aplicación maneja autos. De cada auto, nos interesa: un nombre que lo identifica, el año de fabricación y su valor (en dólares), lo que será un `double`. Hay un método que nos permite el valor de un auto en un determinado año teniendo en cuenta la depreciación (los autos se deprecian un 5% a partir del año de fabricación).
- La aplicación Web nos permitirá crear un nuevo auto y obtener el valor de los autos de una determinada marca en un determinado año (contando la depreciación). Fíjense que esto son dos páginas Web.

Haremos este ejercicio a partes

- 1. Hagan la capa de dominio: la clase Auto.
- 2. Hagan la JSP de obtener el valor de los autos en un determinado año.
- 3. Hagan la capa de negocio de obtener el valor de los autos en un determinado año.
- 4. Hagan la capa de datos de obtener el valor de los autos en un determinado año (mirar el ejemplo anterior).
- 5. Creen la BD, despliegen y ejecuten.
- 6. Repitan lo mismo con crear un nuevo auto.

Sobre el profesor

Dr. Vicent-Ramon Palasí Lallana.

- **Consultas y dudas a:**
  Teléfono: 275-4254.
  Fax: 263-3948.
  E-mail: vpalasi@aurumsol.com
  vpalasi@ufg.edu.sv
  Web: www.aurumsol.com