

Explotación desde aplicaciones externas

*Máster en Business Intelligence e
Innovación Tecnológica*

Índice de contenidos

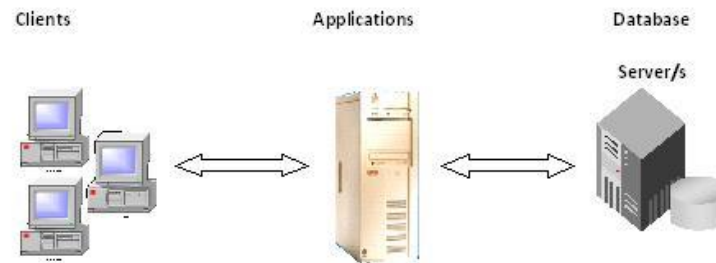
1. Introducción.
2. Conexión mediante drivers (ODBC y JDBC).
3. Conexión desde Python
4. Introducción a la persistencia, Java e Hibernate.
5. API's de servicios.

Integración de bases de datos en aplicaciones

Como hemos visto hasta ahora, los SGBD son una herramienta para almacenar, mantener y explotar la información que contienen. No obstante, no muestran una interfaz amigable para acceder a esa información ni ser explotada por un usuario medio.

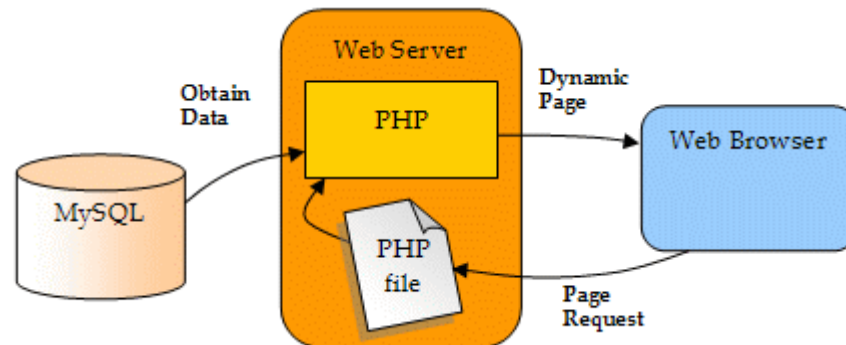
Por este motivo, vemos como la mayoría de bases de datos de la actualidad vienen integradas en una aplicación que conforma una arquitectura más compleja. Por lo tanto, aunque una base de datos contenga la información de un sistema ésta será explotada desde una aplicación codificada en un lenguaje de programación más adecuado para sus propósitos: C, Java, VB, Php o cualquier otro, la mayoría de los lenguajes de programación actuales tienen sus herramientas para gestionar dichas conexiones.

Estas aplicaciones, se conectarán como clientes a los servidores de la base de datos.



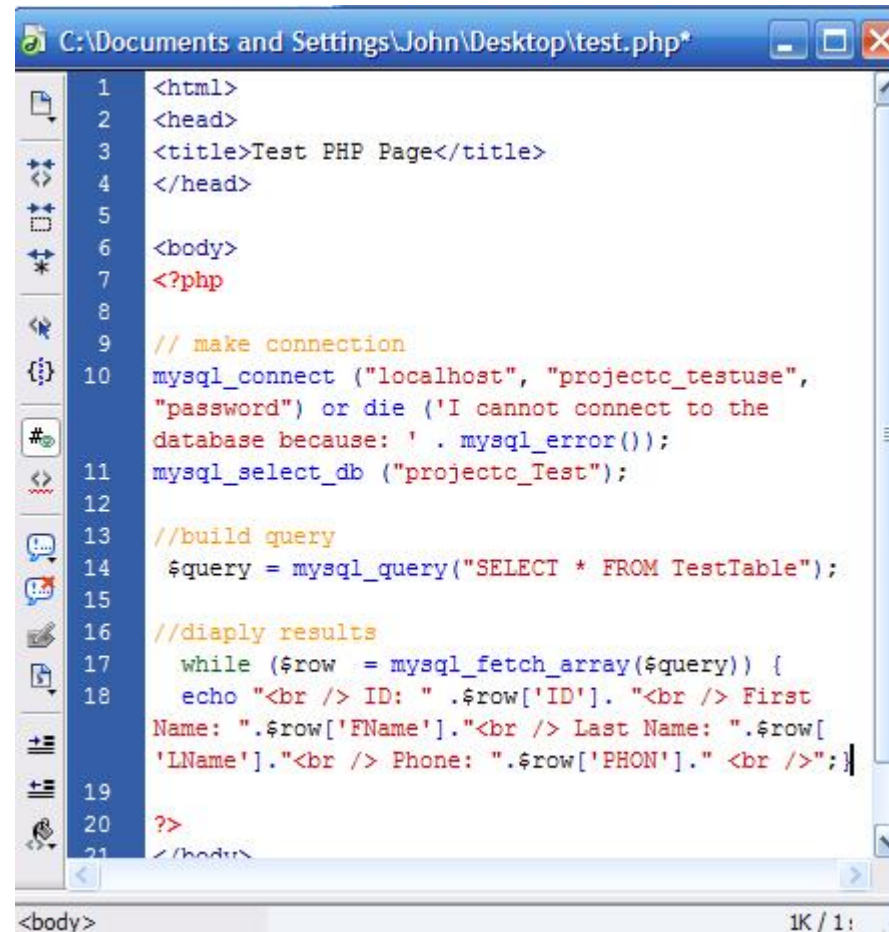
Ejemplo: conexión desde PHP

Un ejemplo sencillo de dicha explotación es la conexión desde un cliente PHP. Este escenario podría ser el de una aplicación web sencilla como la de una tienda on-line (por ejemplo Prestashop), un blog (por ejemplo Wordpress) o incluso aplicaciones más avanzadas como por ejemplo Facebook.



Sería un ejemplo de las llamadas aplicaciones LAMP, aplicaciones web definidas por su stack tecnológico: Linux, Apache, Mysql, PHP, de amplio uso en el mundo actual.

Ejemplo: conexión desde PHP



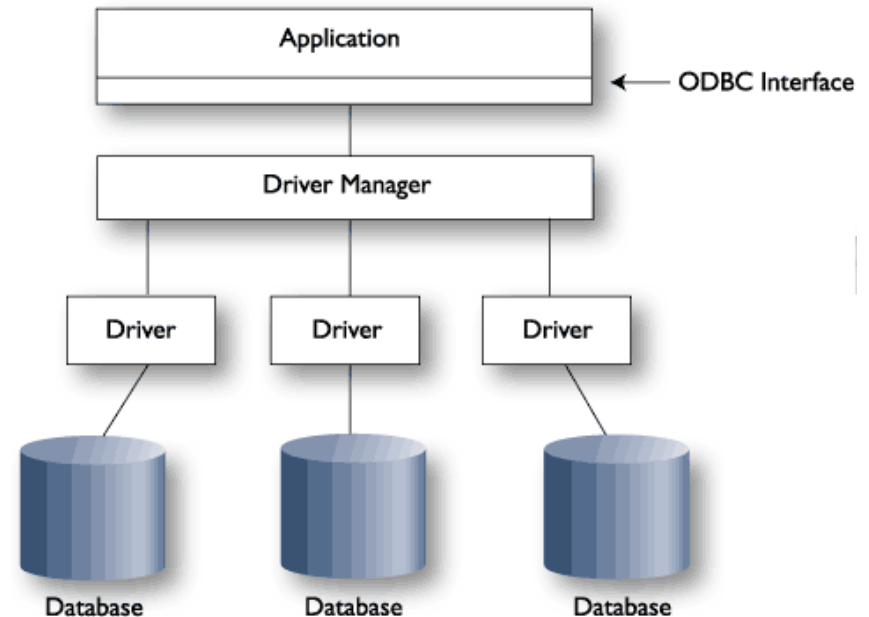
The screenshot shows a Windows Notepad window titled "C:\Documents and Settings\John\Desktop\test.php*". The window contains a PHP script with the following code:

```
1 <html>
2 <head>
3 <title>Test PHP Page</title>
4 </head>
5
6 <body>
7 <?php
8
9 // make connection
10 mysql_connect ("localhost", "projectc_testuse",
11 "password") or die ('I cannot connect to the
12 database because: ' . mysql_error());
13 mysql_select_db ("projectc_Test");
14
15 //build query
16 $query = mysql_query("SELECT * FROM TestTable");
17
18 //display results
19 while ($row = mysql_fetch_array($query)) {
20     echo "<br /> ID: " . $row['ID'] . "<br /> First
21     Name: " . $row['FName'] . "<br /> Last Name: " . $row[
22     'LName'] . "<br /> Phone: " . $row['PHON'] . "<br />";
23 }
24
25 ?>
26 </body>
```

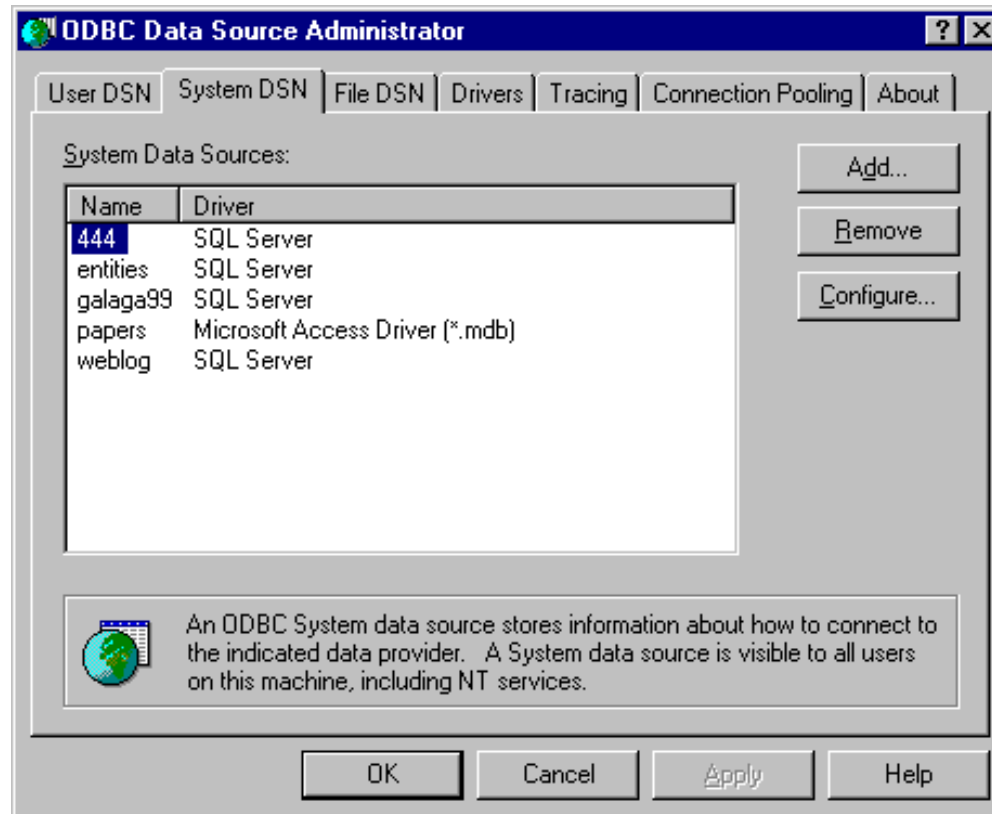
The status bar at the bottom of the window shows "<body>" and "1K / 1:".

ODBC (Open DataBase Connectivity)

- ODBC es un estándar de acceso a bases de datos definido por SQL Acces Group en 1992.
- El objetivo principal es hacer posible el acceso a la información desde cualquier aplicación, sea cual sea el SGBD destino.
- ODBC inserta una capa intermedia entre la aplicación y el SGBD llamada CLI, la interfaz de cliente SQL, que es quién traduce las sentencias que efectúa la aplicación en comandos que el SGBD es capaz de entender.
- El CLI se comunicará con las BDD a través de DataSources, que configuran el acceso mediante drivers específicos.



ODBC (Open DataBase Connectivity)



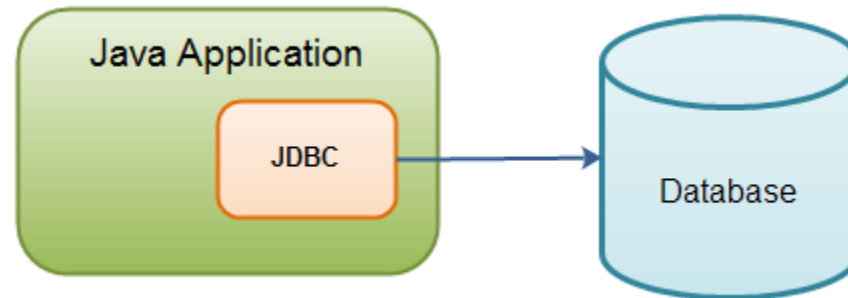
ODBC (Open DataBase Connectivity)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Type_One
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //Load Driver
            Connection con = DriverManager.getConnection("jdbc:odbc:HOD_DATA"); //Create
            Connection with Data Source Name : HOD_DATA
            Statement s = con.createStatement(); // Create Statement
            String query = "select * from Data"; // Create Query
            s.execute(query); // Execute Query
            ResultSet rs = s.getResultSet(); //return the data from Statement into ResultSet
            while(rs.next()) // Retrieve data from ResultSet
            {
                System.out.print("Serial number : "+rs.getString(1)); //1st column of Table from database
                System.out.print(" , Name : "+rs.getString(2)); //2nd column of Table
                System.out.print(" , City : "+rs.getString(3)); //3rd column of Table
                System.out.println(" and Age : "+rs.getString(4)); //4th column of Table
            }
        }
    }
}
```


JDBC (Java DataBase Connectivity)

- JDBC es una API de acceso a datos proporcionada por Java como símil propio a ODBC basándose en ésta.
- Del mismo modo que ODBC, proporciona una serie de drivers para acceder a cualquier SGBD desde cualquier aplicación Java.
- Desarrollado en y para ésta tecnología, los drivers y elementos son proporcionados mediante *interfaces* Java.



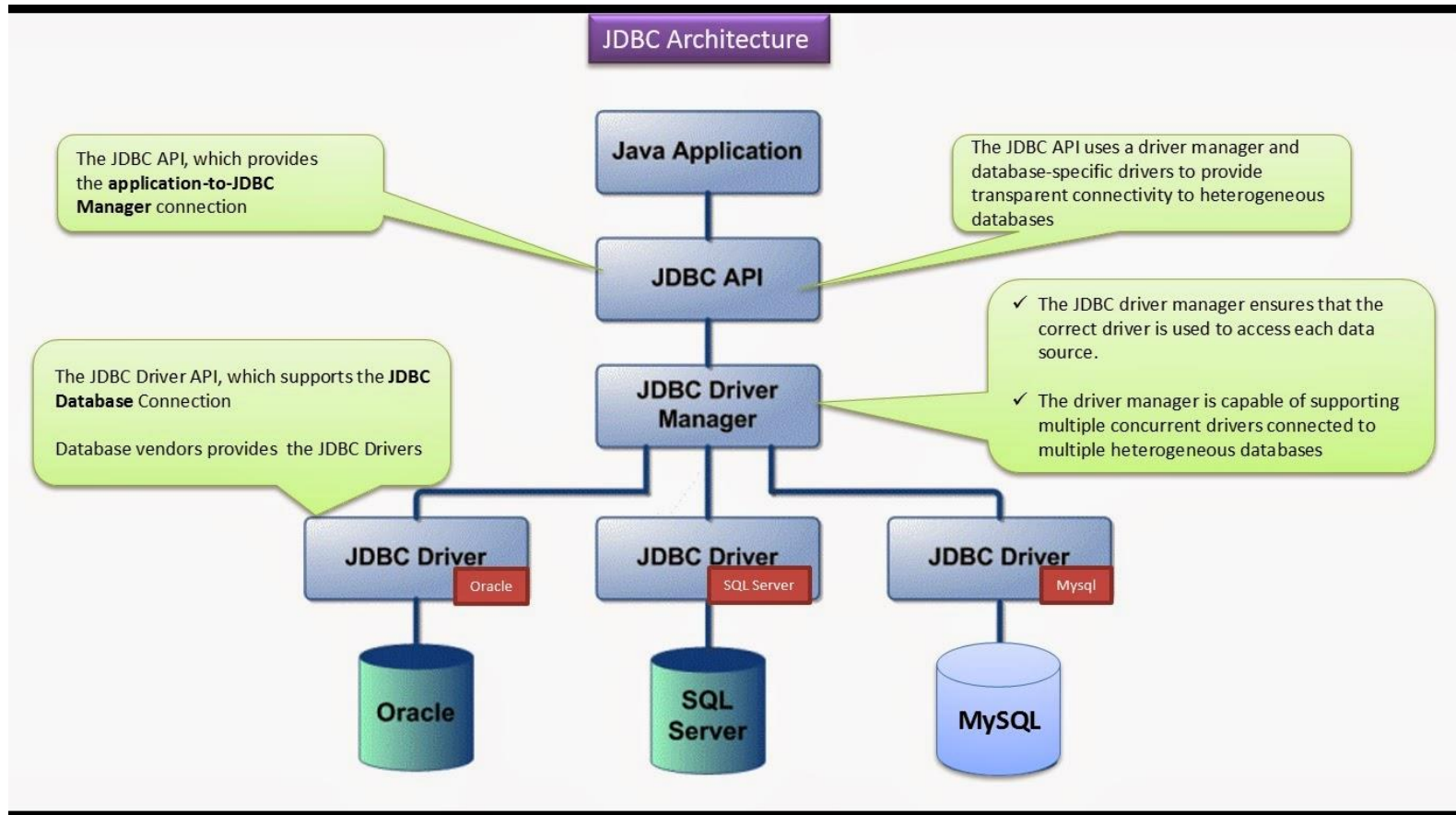
JDBC (Java DataBase Connectivity)

Elementos:

Clase	Descripción
DriverManager	Para cargar un driver
Connection	Para establecer conexiones con las bases de datos
Statement	Para ejecutar sentencias SQL y enviarlas a las BBDD
PreparedStatement	La ruta de ejecución está predeterminada en el servidor de base de datos que le permite ser ejecutado varias veces
ResultSet	Para almacenar el resultado de la consulta

JDBC (Java DataBase Connectivity)

Arquitectura:



JDBC (Java DataBase Connectivity)

Ejemplo:

```
//STEP 2: Register JDBC driver
Class.forName("com.mysql.jdbc.Driver");

//STEP 3: Open a connection
System.out.println("Connecting to database...");
conn = DriverManager.getConnection(DB_URL,USER,PASS);

//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);

//STEP 5: Extract data from result set
while(rs.next()){
    //Retrieve by column name
    int id = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");

    //Display values
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
}
```



ejemplo jdbc.java

JDBC vs ODBC:

- **Multithread:** JDBC es multithread, ODBC no (al menos thread safe)
- **Flexibilidad:** ODBC es una tecnología específica de windows - JDBC es específico para Java y soportado por cualquier OS que soporte Java.
- **Power :** se puede realizar lo mismo en JDBC que en ODBC, en cualquier plataforma.
- **Lenguaje:** ODBC es procedural y independiente del lenguaje - JDBC es orientado a objetos y dependiente del lenguaje (específico de Java).
- **Tolerancia a Alta Carga:** JDBC más rápido - ODBC más lento.
- **ODBC limitaciones:** es una API relcional y sólo puede trabajar con tipos de datos que pueden ser expresados en un formato bidimensional (no trabajará con tipos de datos como Blob, espaciales, etc.).
- **API:** La API JDBC es una interface natural bajo ODBC, por lo tanto JDBC retiene las funcionalidades básicas de ODBC.

Introducción

Python, como casi cualquier lenguaje de programación, permite conectarse a cualquier base de datos para trabajar con ella y evidentemente lo hace con MySQL. De hecho, la pareja Python + MySQL es muy extendida por ejemplo en tecnologías web.

Para ello, primero deberemos instalar el módulo de comunicaciones con MySQL, que en este caso actuará como driver.

```
sudo apt-get install python-mysqldb
```

Conexiones

Para trabajar con MySQL desde una aplicación Python tenemos que inicializar primero conexiones desde el código:

```
bd = MySQLdb.connect(host,user,password,database)
```

Utilizaremos las conexiones para realizar las operaciones que necesitemos. Una vez acabado, deberemos cerrarlas para liberarlas dentro del motor de MySQL:

```
bd.close()
```


Cursores

Un cursor es una estructura de control que se usa para recorrer (y eventualmente procesar) los records de un result set.

Debemos inicializarlo desde una conexión:

```
cursor = bd.cursor()
```

Una vez inicializado, con él ejecutaremos consultas SQL:

```
cursor.execute("select * from empleados")
```

Y después de ejecutar una consulta, a través de él recuperaremos los resultados:

```
registro = cursor.fetchone()  
resultados = cursor.fetchall()
```



Recuperar el primer resultado



Recuperar todos los resultados

Cursores

Si una consulta devuelve más de un campo, la estructura de datos que devuelve el fetch será de tipo array. Para obtener los valores, deberemos recuperarlos de dicha estructura:

```
nombre = registro[0]  
apellido = registro[1]  
edad = registro[2]  
sexo = registro[3]  
salario = registro[4]
```

*“select nombre, apellido, edad,
sexo, salario from empleados”*

Si recuperamos más de un resultado, por ejemplo con el fetchall, deberemos procesar cada elemento de una lista. Por ejemplo:

```
for registro in resultados:  
    nombre = registro[0]  
    apellido = registro[1]
```

Transacciones

Python permite gestionar la transaccionalidad de una o varias operaciones directamente con commit y rollback.







Por ejemplo:

```
try:
    cursor.execute(sql)
    bd.commit()
except:
    bd.rollback()
```

El try inicializa un bloque de errores controlados: siempre que se produzca uno se ejecutarán las operaciones especificadas bajo el except. Por tanto, si el execute va bien ejecutaremos un commit y si va mal un rollback.

Ejemplos

Está disponible un set de ejemplos en el portal github:

 <code>ejemplo_create.py</code>	Rename ejemploCREATE.py to ejemplo_create.py
 <code>ejemplo_delete.py</code>	Rename ejemploDELETE.py to ejemplo_delete.py
 <code>ejemplo_insert.py</code>	Rename ejemploINSERT.py to ejemplo_insert.py
 <code>ejemplo_select.py</code>	Rename ejemploSELECT.py to ejemplo_select.py
 <code>ejemplo_update.py</code>	Rename ejemploUPDATE.py to ejemplo_update.py
 <code>intro_sql.py</code>	Rename introSQL.py to intro_sql.py

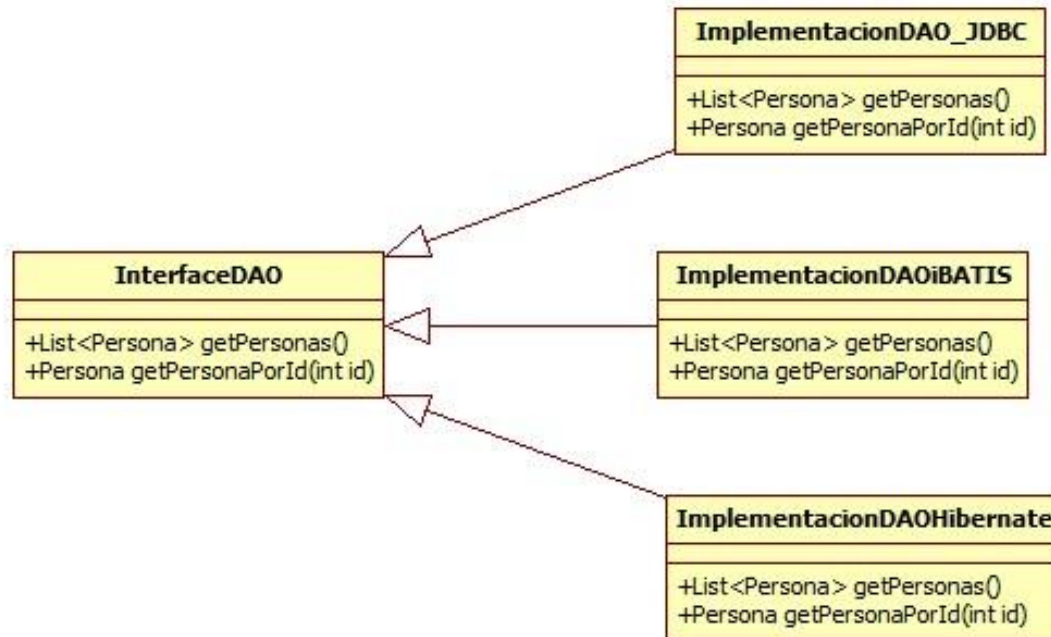
<https://github.com/JoanPinol/EAE>

Introducción a ORM

- Una de las tendencias actuales en el diseño de aplicaciones es la separación multicapa, de forma que se proporciona una mayor independencia tecnológica y adaptabilidad.
- Gracias a la aparición de dichas capas aparece el patrón DAO (Data Access Object), que proporciona objetos Java capaces de realizar el acceso a la BDD y posterior mapeo de los campos en sus atributos.
- A partir de los objetos DAO aparece el concepto de persistencia: los datos que contiene una instancia de un objeto DAO deben persistir a la vida de ejecución de la aplicación.
- DAO, por tanto, es una interfaz entre el acceso a datos (y persistencia) y el negocio que permite desacoplar la aplicación.

Introducción a ORM

- El patrón DAO es independiente de la tecnología, simplemente es un patrón de desarrollo.



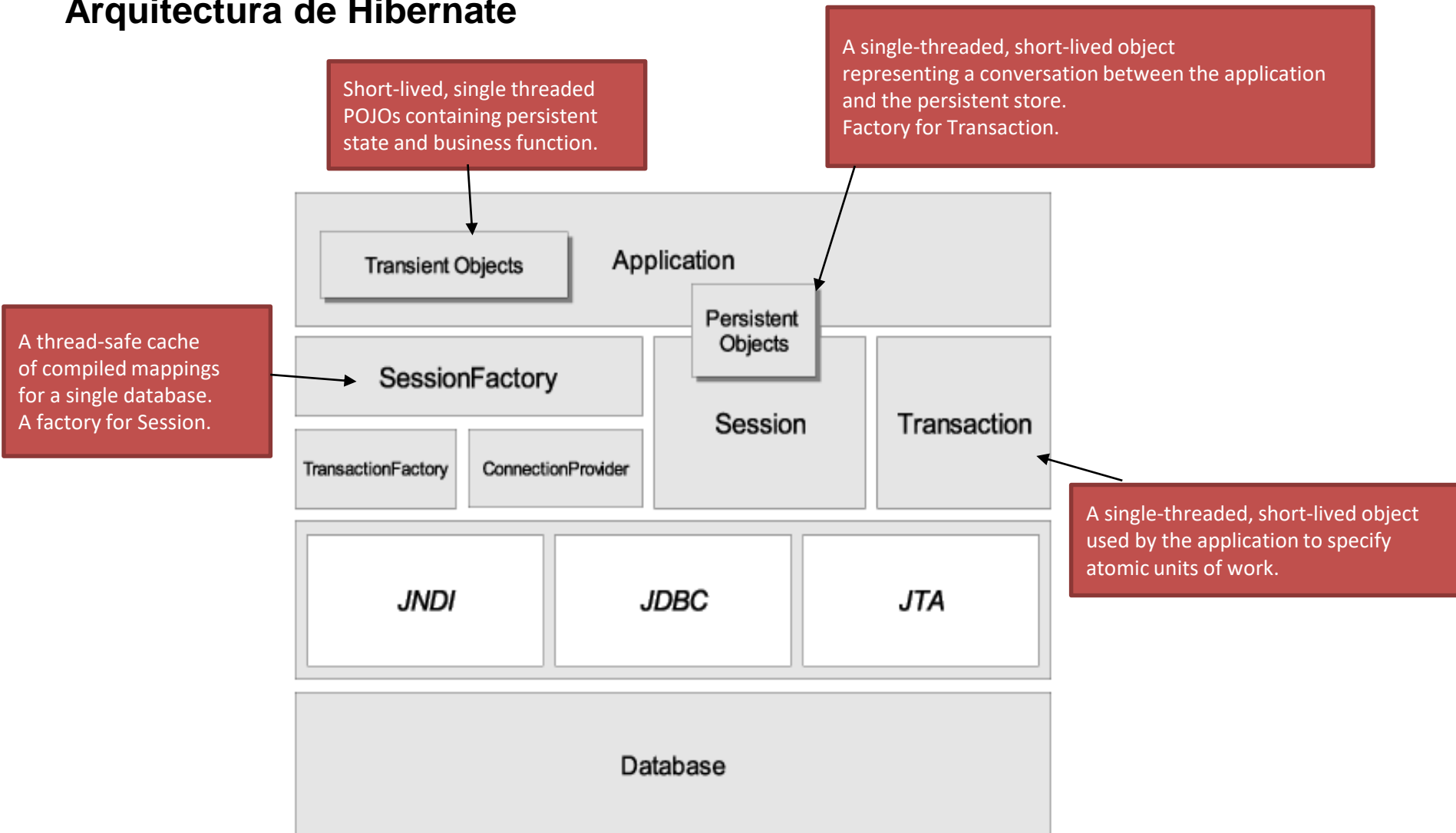
Introducción a ORM

- A partir de lo anterior aparece el concepto ORM (Object-Relational Mapping): una herramienta que permita traducir los objetos en los elementos relacionales y viceversa.
- ORM proporciona:
 - Una API para realizar las operaciones CRUD sobre los objetos de la aplicación.
 - Un mecanismo de mapeo entre los objetos y las tablas.
 - Una tecnología transaccional para la asociación entre objetos y tablas.
- Sus principales características:
 - Detección de cambios y gestión de transacciones.
 - Mecanismos de detección de pérdida de información o actualizaciones.
 - Mapping specification.
 - Lenguaje propio de consultas.
 - Gestión de persistencia.
 - Elementos propios de la POO: herencia, sobrecarga, etc.
 - Estrategias de carga de la información
 - Gestión de caché propia.
 - Independencia del SGBD.
- Algunos ORM: Hibernate (java), JPA (java), Doctrine (php), peewee (python), ADO.NET Entity Framework (C#).

Introducción a Hibernate

- Hibernate es una implementación de ORM open-source basada en tecnología Java.
- Se trata de una re-implementación basada en tecnología JPA.
- Los mapeos se realizan mediante configuración en ficheros xml o anotaciones Java
- Es posible utilizarlo en todas las aplicaciones Java, no sólo J2EE.
- Los objetos utilizados en el mapeo son simples objetos POJO (Plain Ordinary Java Object, similares a Java Beans)

Arquitectura de Hibernate



Configuración de Hibernate

- Fichero hibernate.cfg.xml
- Definir SGBD y parámetros de configuración.
- Definir conexión a BDD.
- Definir driver y modo de conexión.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.bytecode.use_reflection_optimizer">false</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.password">password</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mkyong</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="show_sql">true</property>
    <mapping resource="com/mkyong/common/Stock.hbm.xml"></mapping>
  </session-factory>
</hibernate-configuration>
```

SessionFactory

- Genera la conexión a partir de hibernate.cfg.xml

```
package com.mkyong.persistence;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory();
        }
        catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        // Close caches and connection pools
        getSessionFactory().close();
    }
}
```

Creando el objeto POJO

```
package com.mkyong.common;

public class Stock implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private Integer stockId;
    private String stockCode;
    private String stockName;

    public Stock() {
    }
    public Stock(String stockCode, String stockName) {
        this.stockCode = stockCode;
        this.stockName = stockName;
    }
    public Integer getStockId() {
        return this.stockId;
    }
    public void setStockId(Integer stockId) {
        this.stockId = stockId;
    }
    public String getStockCode() {
        return this.stockCode;
    }
    public void setStockCode(String stockCode) {
        this.stockCode = stockCode;
    }
}
```

Creando el objeto POJO

```
public String getStockName() {  
    return this.stockName;  
}  
public void setStockName(String stockName) {  
    this.stockName = stockName;  
}  
}
```

Fichero de Mapeo para el POJO

```
<?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.mkyong.common.Stock" table="stock" catalog="mkyong">
    <id name="stockId" type="java.lang.Integer">
      <column name="STOCK_ID" />
      <generator class="identity" />
    </id>
    <property name="stockCode" type="string">
      <column name="STOCK_CODE" length="10" not-null="true" unique="true" />
    </property>
    <property name="stockName" type="string">
      <column name="STOCK_NAME" length="20" not-null="true" unique="true" />
    </property>
  </class>
</hibernate-mapping>
```


Ejecución

```
package com.mkyong.common;

import org.hibernate.Session;
import com.mkyong.persistence.HibernateUtil;

public class App
{
    public static void main( String[] args )
    {
        System.out.println("Maven + Hibernate + MySQL");
        Session session = HibernateUtil.getSessionFactory().openSession();

        session.beginTransaction();
        Stock stock = new Stock();

        stock.setStockCode("4715");
        stock.setStockName("GENM");

        session.save(stock);
        session.getTransaction().commit();
    }
}
```

Características del objeto POJO

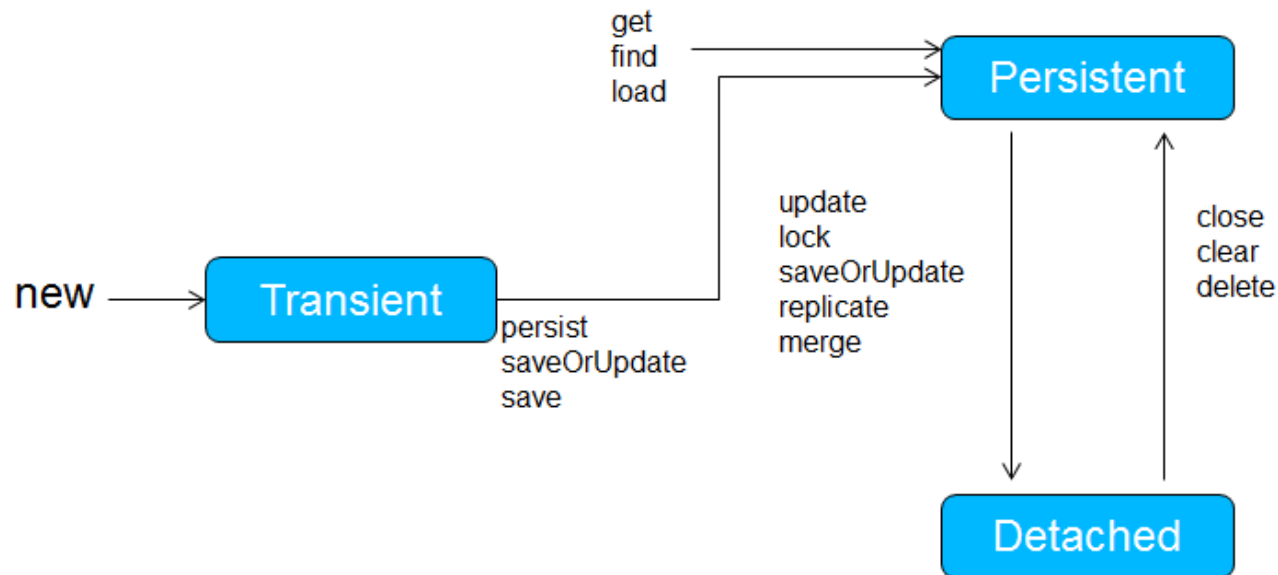
- Es recomendable implementar la interfaz Serializable
- Debe incluir constructor por defecto sin parámetros
- No declarar la clase ni sus métodos como final.
- Debe tener un atributo de tipo int para la id o pk de la tabla.
- Los atributos de la clase deben corresponderse con las columnas con sus respectivos getters y setters.

Estados del objeto POJO

Transient: El objeto se ha instanciado mediante new pero no se ha asociado a ninguna sesión. No tiene ni identificador ni persistencia en la BDD.

Persistent: Una instancia persistente con representación a una BDD, está asociado a una sesión de HB y sincronizado con el objeto de la BDD.

Detached: Instancia que ha estado persistente pero se ha cerrado la sesión con la BDD. La referencia y la id son válidos y se pueden utilizar en una nueva sesión.



HQL

- HQL es el lenguaje de consultas propio de Hibernate.
- Gracias a él se desacopla el código de las consultas del SGBD, haciéndolo transparente a la interpretación propia de SQL-92 del mismo y convirtiendo la aplicación en transparente a todos ellos.
- Es un lenguaje de consulta orientado a objetos.
- Permite utilizar funciones: avg(), sum(), min(), max(), count(), distinct(),...
- Permite utilizar subconsultas y las cláusulas de ordenación y agrupación.

Ejemplo básico:

```
from Users
```

Ejemplo estándar:

```
Select nombre,nif from Usuario usr  
where usr.idusuario=13
```

HQL

Se utiliza de forma sencilla mediante la clase Query, que además permite incluir parámetros.

```
begin();  
Query q = getSession().createQuery("from Categoria");  
List lista = q.list();  
commit();  
return lista;
```

```
begin();  
Query q = getSession().createQuery("from Usuario where nombre = :username");  
q.setString("username",username);  
Usuario usr = (Usuario)q.uniqueResult();  
commit();  
return usr;
```

HQL

Es posible almacenar consultas en los ficheros de mapping

```
<query name="Product.findAllProductsByProductName">
<![CDATA[from Product pro where pro.name = :productName]]>
</query>

<query name="Product.findAllProductsByProductPrice">
<![CDATA[from Product pro where pro.price between :minPrice and :maxPrice]]>
</query>
```

Y posteriormente utilizarlas desde el código

```
Query queryProductsByName =
session.createNamedQuery("Product.findAllProductsByProductName");
//set dynamic data for query
queryProductsByName.setString("productName", productName);
//execute query and get results
List products = queryProductsByName.list();
```

Introducción a Hibernate

- Tutorial utilizado:

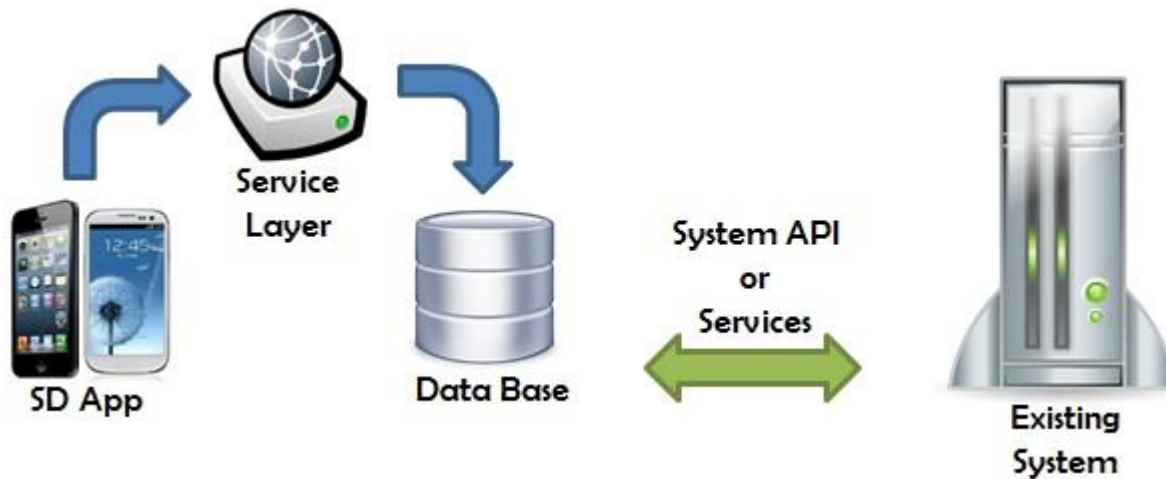
<http://www.mkyong.com/hibernate/quick-start-maven-hibernate-mysql-example/>

- Para ampliar conocimientos de Hibernate existen numerosos tutoriales. Por ejemplo:

<https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/tutorial.html>

Introducción

- Con el auge de las aplicaciones distribuidas, aparece un nuevo método de acceso a datos: API's de servicios.
- Basándose en tecnologías como WebServices, son aplicaciones que desplegadas en un servidor proporcionan interfaces de acceso a datos a otras aplicaciones.
- En una primer momento, la tecnología que apareció proponía intercambio de información mediante XML y tecnología SOAP.



Un ejemplo son las APIs de servicios para aplicaciones móviles

Introducción

- En esta tónica aparecen distintas tecnologías, como por ejemplo los microservicios, que además proporcionan interfaces reaprovechables para distintos canales.
- Así, proporcionan acceso a datos a aplicaciones web, aplicaciones nativas para smartphones, etc. bajo un único punto de acceso.
- Además, se abstrae el mensaje intercambiado haciéndolo directamente en llamadas http (igual que un browser): aparecen las API Rest y mensajes JSON.

Microservice Architecture

