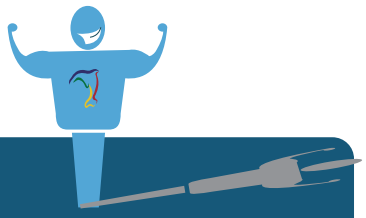
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

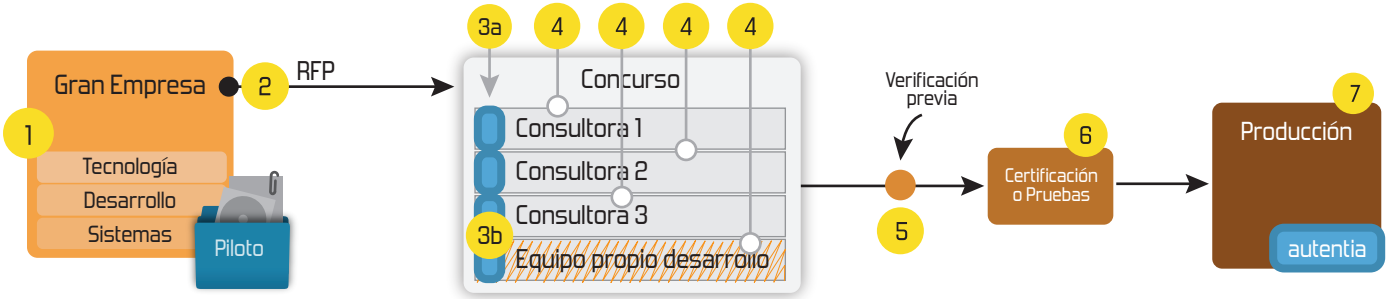
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Tutorial desarrollado por: Francisco Javier Martínez Páez

Puedes encontrarme en Autentia
Somos expertos en Java/J2EE
Contacta en info@autentia.com

Descargar este documento en formato PDF introjdbc.pdf

1. **¿ Qué es JDBC ?**

Evidentemente, para muchos, la respuesta a esta pregunta es ya sabida. Sin embargo, nos llegan gran cantidad de cuestiones acerca de este tema, y es por eso que hemos decidido hacer una serie de tutoriales que cubran la mayor parte de los aspectos y características de este estándar definido por SUN. Este primer tutorial es puramente teórico, más adelante haremos otros tutoriales referentes a este tema, donde cubriremos algunos aspectos del API de JDBC menos conocidos, y también uno final donde mostraremos algunas de las últimas novedades incluidas en el API de Java5.

Bueno...pues como diría un anglosajón "Let`s get down to work" o manos a la obra:

JDBC es un **API** (Application programming interface) que describe o define una librería estándar para a**cceso a fuentes de datos**, principalmente orientado a Bases de Datos relacionales que usan SQL (Structured Query Language). JDBC no sólo provee un interfaz para acceso a motores de bases de datos, sino que también **define una arquitectura estándar**, para que los fabricantes puedan crear los drivers que permitan a las aplicaciones java el acceso a los datos.

Sun define JDBC como: *"The JDBC API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases."*

1. **Filosofía y Objetivos de JDBC**

Cuando SUN se puso a trabajar en este tema, decidió seguir una serie de normas a seguir para la definición del interfaz, y que han condicionado en gran manera el resultado final. Algunas de estas características son:

- **API A NIVEL SQL**. JDBC es un API de bajo nivel, es decir, que está orientado a permitir ejecutar comandos SQL directamente, y procesar los resultados obtenidos. Esto supone que será tarea del programador crear APIs de más alto nivel apoyándose directamente sobre JDBC.
- **COMPATIBLE CON SQL.** Cada motor de Base de Datos implementa una amplia variedad de comandos SQL, y muchos de ellos no tienen porque ser compatibles con el resto de motores de Base de Datos. JDBC, para solventar este problema de incompatibilidad, ha tomado la siguiente posición
  - JDBC permite que cualquier comando SQL pueda ser pasado al driver directamente, con lo que una aplicación Java puede hacer uso de toda la funcionalidad que provea el motor de Base de Datos, con el riesgo de que esto pueda producir errores o no en función del motor de Base de Datos.
  - Con el objetivo de conseguir que un driver sea compatible con SQL (SQL compliant), se obliga a que al menos, el driver cumpla el Estándar ANSI SQL 92.
- **JDBC debe ser utilizable sobre cualquier otro API de acceso a Bases de Datos**, o más en particular ODBC (Open Database Connectivity)
- **JDBC debe proveer un interfaz homogéneo** al resto de APIs de Java.
- **JDBC debe ser un API simple**, y desde ahí, ir creciendo.
- **JDBC debe ser fuertemente tipado, y siempre que sea posible de manera estática**, es decir, en tiempo de compilación, para evitar errores en tiempo de ejecución.
- **JDBC debe mantener los casos comunes de acceso a Base de Datos lo más sencillo posible**:
  - Mantener la sencillez en los casos más comunes (SELECT, INSERT, DELETE y UPDATE)
  - Hacer realizables los casos menos comunes: Invocación de procedimientos almacenados...
- **Crear múltiples métodos para múltiple funcionalidad**. JDBC ha preferido incluir gran cantidad de métodos, en lugar de hacer métodos complejos con gran cantidad de parámetros.

3. **Procedimiento de Conexión y acceso a datos con JDBC.**

- **Consideraciones previas.**

El proceso de acceso a una Base de Datos a través de JDBC, exige dar una serie de pasos previos antes de crear la conexión al motor de Base de Datos. El primer paso es determinar el entorno en el que el proyecto va a ser instalado, y más en concreto, que parámetros del entorno afectan directamente a JDBC:

¿ Qué motor de Base de Datos vamos a usar ?

Debemos considerar las características específicas de una base de datos, como por ejemplo, como mapear los tipos de datos SQL a Java.

¿ Qué driver vamos a usar ?

Es probable encontrarnos varios drivers distintos para la misma fuente de datos. Debemos saber detectar cual es el driver más adecuado para nuestra aplicación, por ejemplo, si elegimos un driver ODBC/JDBC, tendremos más flexibilidad para elegir distintas fuentes de datos, pero si por ejemplo trabajamos con una Base de Datos Oracle, un driver JDBC diseñado específicamente para esta base de datos será mucho más eficiente.

¿ Donde estará localizado el driver ?

En función de donde se encuentre el driver físicamente, debemos considerar aspectos de rendimiento y seguridad. Por ejemplo, si cargamos el driver desde un servidor remoto tendremos que considerar aspectos sobre seguridad de Java.

- **Procedimiento de conexión.**

El procedimiento de conexión con el controlador de la base de datos, independientemente de la arquitectura es siempre muy similar.

1. Cargar el driver. Cualquier driver JDBC, independientemente del tipo debe implementar el interfaz java.sql.Driver. La carga del driver se puede realizar de dos maneras distintas:

- Definiendo los drivers en la variable ***sql.driver*** (variable que mantiene todos las clases de los drivers separados por comas) Cuando la clase DriverManager se inicializa, busca esta propiedad en el sistema.
- El programador puede forzar la carga de un driver específico, usando el método Class.forName(driver).

2. Registro del driver. Independientemente de la forma de carga del driver que llevemos a cabo, será responsabilidad de cada driver registrarse a sí mismo, usando el método DriverManager.registerDriver. Esto permite a la clase DriverManager, usar cada driver para crear conexiones con el controlador de Base de Datos. Por motivos de seguridad, la capa que gestiona JDBC, controlará en todo momento que driver es el que se está usando, y cuando se realicen conexiones, sólo se podrán usar drivers que estén en el sistema local de ficheros o que usen el mismo ClassLoader que el código que está intentando crear la conexión.

En el primero de los casos (a través de la propiedad sql.driver), JDBC usará el primer driver que permita conectarse correctamente a la Base de Datos.

3. Crear una conexión. El objetivo es conseguir un objeto del tipo java.sql.Connection a través del método DriverManager.getConnection(String url). La capa de gestión, cuando este método es invocado, tratará de encontrar un driver adecuado para conectar a la base de datos especificada en la URL, intentándolo por el orden especificado en la variable sql.driver. Cada driver debería examinar si ellos proveen el "subprotocolo" que especifica la URL. (Ver anexo)

3. **Tipos de conectores (drivers) JDBC**

Los conectores o drivers JDBC, se pueden dividir en cuatro tipos principalmente:

- **Tipo 1. JDBC-ODBC bridge más driver ODBC: "BRIDGE"**

Permite el acceso a Base de Datos JDBC mediante un driver ODBC. Cada máquina cliente que use el puente, debe tener librerías clientes de ODBC(dll propias del S.O)

- Ventajas: Buena forma de aprender JDBC. También puede ser buena idea usarlo, en sistemas donde cada máquina cliente tenga ya instalado los drivers ODBC. También es posible que sea la única forma de acceder a ciertos motores de Bases de Datos.
- Inconvenientes: No es buena idea usar esta solución para aplicaciones que exijan un gran rendimiento, ya que la transformación JDBC-ODBC es costosa. Tampoco es buena solución para aplicaciones con alto nivel de escalabilidad.

- **Tipo 2. Driver Java parciales: "NATIVE"**

Traducen las llamadas al API de JDBC Java en llamadas propias del motor de Base de Datos (Oracle, Informix...). Al igual que el tipo anterior, exige en las máquinas clientes código binario propio del cliente de la Base de datos específica y del sistema operativo

- Ventajas: Mejor rendimiento que el anterior. Quizá puede ser buena solución para entornos controlados como intranets. Ejemplo OCI oracle.
- Inconvenientes: Principalmente la escalabilidad, ya que estos drivers exigen que en la máquina cliente librerías del cliente de la Base de Datos.

- **Tipo 3. Driver JDBC a través de Middleware: "NETWORK"**

Traduce las llamadas al API JDBC en llamadas propias del protocolo específico del broker. Éste se encargará de traducirlas de nuevo en sentencias propias del motor de Base de Datos de cada caso.

- Ventajas: Buena solución cuando necesitamos acceder a Bases de Datos distintas y se quiere usar un único driver JDBC para acceder a las mismas. Al residir la traducción en el servidor del middleware, los clientes no necesitan librerías específicas, tan solo el driver.
- Inconvenientes: La desventaja principal reside en la configuración del servidor donde se encuentra el middleware. Necesitará librerías específicas para cada motor de base de datos distinto, etc.

- **Tipo 4: Driver java puro (acceso directo a Base de Datos): "THIN".**

Convierte o traduce las llamadas al API JDBC en llamadas al protocolo de red usado por el motor de bases de datos, lo que en realidad es una invocación directa al motor de bases de datos.

- Ventajas: 100 % portable. Buen rendimiento. El cliente sólo necesita el driver.
- Inconvenientes: Al ser independiente de la plataforma, no aprovecha las características específicas del S.O

4. **Instalación de JDBC**

El proceso de instalación de JDBC se podría dividir en tres partes distintas:

- **Descargar el API de JDBC:**

JDBC viene incluido en el corazón de la plataforma Java, tanto en la J2SE como en la J2EE.

http://java.sun.com/products/jdbc/download.html

Versión actual JDBC 1.3 (incluida en J2SE 1.4)

Versión en borrador JDBC 1.4

- **Instalar el driver en la máquina cliente (máquina que crea las conexiones)**

La instalación del driver dependerá en gran medida del tipo de driver que hayamos seleccionado.

Para los tipo 1 y 2, además de las librerías del driver, necesitaremos ciertas librerías propias del cliente del motor de base de datos que estemos usando y quizá ficheros de configuración...

Para el tipo 3, en la máquina cliente tan solo necesitaremos el driver. En el servidor donde resida el middleware, necesitaremos drivers específicos de cada motor de base de datos y en su caso, librerías propias del cliente del motor de base de datos.

Para el tipo 4, tan solo necesitaremos el driver.

- **Instalar el controlador o motor de base de datos.**

Evidentemente, necesitaremos tener instalado un motor de base de datos. Esta instalación será distinta en función de la Base de Datos y del S.O donde vaya a correr.

5. **Arquitecturas JDBC**

La arquitectura básica de JDBC (ya la hemos visto) es simple. Una clase llamada DriverManager provee un mecanismo para controlar un conjunto de drivers JDBC. Esta clase intenta cargar los drivers especificados en la propiedad del sistema jdbc.drivers. También podemos cargar un driver explicitamente usando Class.forName(). Durante la carga, el driver intentará registrarse a si mismo usando el método clase DriverManager.registerDriver(). Cuando se invoque al método DriverManager.getConnection(), ésta buscará el primer driver de los registrados que pueda manejar una conexión como la descrita en la URL y retornará un objeto que implemente el interfaz java.sql.Connection.

Sin embargo, en función de la localización de la base de datos, el driver, la aplicación y el protocolo de comunicación usado, nos podemos encontrar distintos escenarios que accedan a Base de Datos a través de JDBC:

1. Aplicaciones standalone
2. Applets comunicando con un servidor Web
3. Aplicaciones y applets comunicando con una base de datos a través de un puente JDBC/ODBC.
4. Aplicaciones accediendo a recursos remotos usando mecanismos como Java RMI
5. etc...

Todos ellos se pueden agrupar en dos tipos distintos de arquitecturas:

- **Arquitecturas JDBC en dos capas.**

La aplicación que accede a la base de datos reside en el mismo lugar que el driver de la base de datos. El driver accederá al servidor donde corra el motor de base de datos.

En este caso, será el driver el encargado de manejar la comunicación a través de la red.

En el ejemplo, una aplicación java corriendo en una máquina cliente que usa el driver también local. Toda la comunicación a través de la red con la base de datos será manejada por el driver de forma transparente a la aplicación Java.

- **Arquitecturas JDBC en tres capas.**

Una aplicación o applet corriendo en una máquina y accediendo a un driver de base de datos situado en otra máquina. Ejemplos de esta situación:

1. Un applet accediendo al driver a través de un Web Server
2. Una aplicación accediendo a un servidor remoto que comunica localmente con el driver
3. Una aplicación comunicando con un servidor de aplicaciones que accede a la base de datos por nosotros.

7. **Anexo:**

JDBC provee un mecanismo para permitir nombrar las bases de datos, para que los programadores puedan especificar a que base de datos desean conectarse. Este sistema debe tener las siguientes características:

- Drivers de diferentes tipos pueden tener diferentes formas de nombrar las bases de datos.
- Este sistema de nombrado debe ser capaz de acoger diversos parámetros de configuración de la red.
- Debería ser capaz de acoger cierto nivel de indirección, para que los nombres pudiesen ser resueltos de alguna manera (DNS...)
- URL. Estas características están ya estandarizadas en el concepto de URL. JDBC recomienda la siguiente estructura de nombrado de bases de datos: jdbc:<subprotocol>:<subname>
  - jdbc: parte fija
  - subprotocol: mecanismo particular se acceso a base de datos
  - subname: dependerá del subprotocolo. JDBC recomienda seguir también la convención de nombrado URL: //hostname:port/subsubname
- El subprotocolo odbc: Este subprotocolo ha sido reservado para aquellos que quieran seguir la convención ODBC para nombrado de bases de datos:

  jdbc:odbc:<data-source-name>[;<attribute-name>=<attribute-value>]
- El API JDBC, también provee la posibilidad de pasar una lista de propiedades para crear la conexión: DriverManager.getConnection(String URL, Properties props); Dos propiedades definidas por convención son
  - user
  - password

Puedes opinar sobre este tutorial aquí

Nuevo servicio de notificaciones

Si deseas que te enviemos un correo electrónico cuando introduzcamos nuevos tutoriales, inserta tu dirección de correo en el siguiente formulario.

Subscribirse a Novedades	
e-mail	
	Enviar

Otros Tutoriales Recomendados (También ver todos)

Nombre Corto	Descripción
Acceso seguro a CVS a través de SSH	Os mostramos como segurizar los accesos a CVS a través de SSH, utilizando herramientas gratuitas
Java en tu movil con J2ME	Os enseñamos como construir una aplicación Java capaz de correr en tu Movil gracias a J2ME
Uso de JNDI, includes y cookies en Servlets	En este tutorial veremos como usar variables de entorno desde JNDI, incluir un servlet en otro (include) y como usar cookies en Servlets
Creación de Webs con Power WebSite Builder	Creación de páginas web sin necesidad de conocimientos HTML, usando la herramienta Power Website Builder
Pool de conexiones a BBDD con struts	Os mostramos como configurar un pool de conexiones a base de datos en vuestras aplicaciones construidas con struts
Generación automática de código JDBC	En este tutorial os enseñamos como, sin conocimiento de JDBC, crear vuestro programas en Java, gracias a JDBCTest.
Pool de Conexiones y Tomcat5	Os mostramos como instalar Tomcat5 en vuestro PC y como ejemplo de uso, configuramos un Pool de Conexiones y lo usamos contra MySQL
Introducción a JDBC	En este tutorial os explicamos los fundamentos teóricos de JDBC
JDBC y MySql	En el tutorial anterior vimos como instalar MySQL en Windows, ahora vamos a ver como acceder desde una aplicación Java.
EJB´s y Orion	Recreación de la guía paso a paso de como crear una aplicación Web con EJB´s y Servlets y su despliegue con ANT sobre Orion