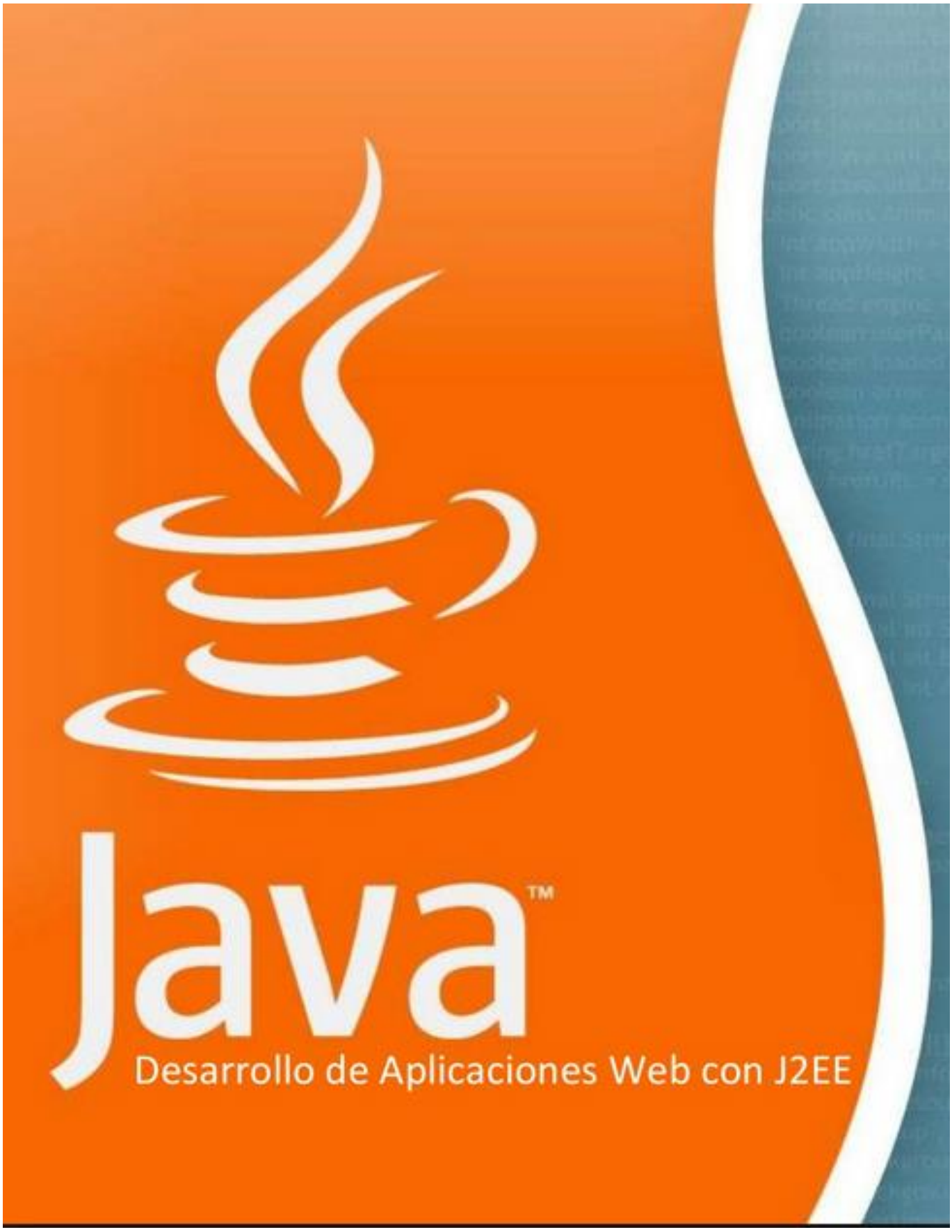

Java™
Desarrollo de Aplicaciones Web con J2EE

3. Acceso a datos

3.1 Acceso a base de datos

Tenemos varias alternativas para realizar la persistencia de la información a la base de datos:

- ✓ Utilizar directamente el API JDBC como lo haríamos en una aplicación Java SE
- ✓ Utilizar un pool de conexiones
- ✓ Utilizar un motor de persistencia como es el API de persistencia de Java JPA, una implementación de este API es Hibernate

3.1.1 JDBC

Utilizar directamente el API JDBC para acceder a la base de datos no es aconsejable. Aparentemente no habría problema: cuando se crea el servlet se crea la conexión con la base de datos, cada vez que hay una petición se realizan las consultas correspondientes y, por último, cuando termina la vida del servlet se cierra la conexión. Sin embargo existen varios problemas:

- ✓ **El tiempo de espera del servlet hasta que se destruye y el tiempo de espera de mysql hasta que cierra la conexión no tienen por qué ser compatibles.** Por ejemplo, nos podemos encontrar con que tras un tiempo de inactividad, mysql cierra la conexión pero el servlet sique cargado. Si antes de destruir el servlet se realiza otra petición tendríamos un error en el acceso a la base de datos.
- ✓ **Podrían darse accesos simultáneos sobre la misma conexión**, lo que produciría errores de acceso a la base de datos. Aunque se sincronice al máximo, con una única sentencia cada vez, al final tenemos una única conexión a la base de datos, y por tanto sólo es posible ejecutar de modo simultáneo una única consulta contra ella. Si varios usuarios están empleando la aplicación, todas esas consultas acabarán en una cola y se irán ejecutando de una en una. Abrir y cerrar conexiones contra una base de datos son operaciones costosas. Por lo tanto no se obtiene un rendimiento óptimo de la aplicación.

Podríamos pensar que una posible opción es crear el objeto Statment y el objeto Connection dentro de los métodos doGet y doPost, de tal modo que se conviertan en variables locales, concentrando todo el trabajo en estos métodos, de forma que para cada petición se realice una nueva conexión, las operaciones contra la base de datos y el cierre de la conexión. Esto está bien para aplicaciones con poco trabajo contra la base de datos, pero cuando se realizan varias peticiones se produce una sobrecarga de trabajo que era la que intentábamos evitar con el caso anterior.

La solución óptima sería no tener una única conexión con la base de datos, sino varias.

Cuando llega una petición http, empleamos una conexión que no se esté usando en ese momento. Cuando una petición http termina, se libere esa conexión. Esto se consigue utilizando un **pool de conexiones**, **de forma que por un lado se gestionen las conexiones a la base de datos y por otro el trabajo contra la base de datos**. En lugar de implementar nuestro propio pool de conexiones que es bastante tedioso podemos utilizar el pool de conexiones que nos ofrezca el servidor de aplicaciones.

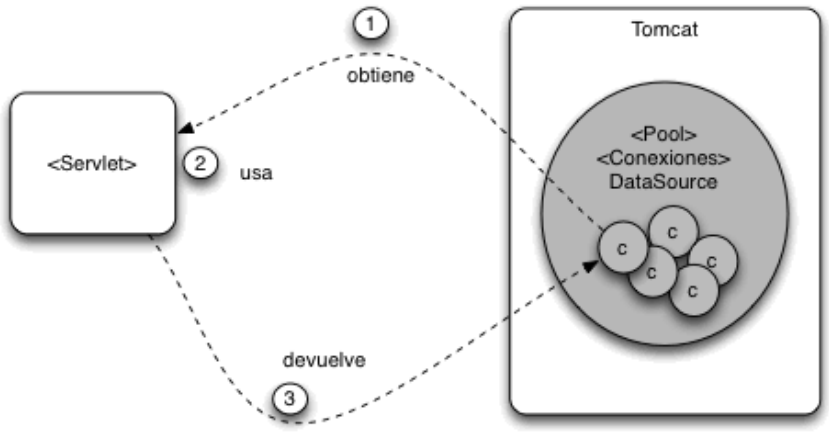
3.1.1.2 Ejemplo conexión por JDBC

```
public ServletJDBC() {
    super();
    // TODO Auto-generated constructor stub
}

/**
 * @see Servlet#init(ServletConfig)
 */
public void init(ServletConfig config) throws ServletException {
    // TODO Auto-generated method stub

    try {
        conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/prueba", "root", "");
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
    response.setContentType("text/plain");
    if (conexion != null)
        response.getWriter().append("Conexión correcta");
    else
        response.getWriter().append("Conexión correcta");
}

@Override
public void destroy() {
    try {
        conexion.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

3.1.2 Pool de Conexiones

La forma más adecuada de utilizar las conexiones a la base de datos dentro de un servidor de aplicaciones Java EE es emplear un pool de conexiones que esté gestionado por el propio servidor. Para ello, empleando la consola de administración del servidor, deberemos configurar un pool de conexiones dentro de él.

En este apartado nos vamos a centrar en el pool de conexiones que ofrece el contenedor de servlets Tomcat. Es importante tener en cuenta que de una versión a otra de Tomcat puede cambiar la configuración del pool de conexiones, por lo que es importante revisar la documentación de la versión de Tomcat que tengamos instalada.



El uso de estas conexiones es durante un tiempo determinado en el cual el Servlet, JSP o cualquier componente la necesitan. Una vez que han terminado de usarla la devuelven al pool y así permiten su reutilización.

Vamos a ver un ejemplo de cómo crear un pool de conexiones con el servidor Apache Tomcat en una aplicación Java EE. Además de configurarlo, deberemos asignarle un **nombre JNDI** (Java Naming and Directory Interface) a este recurso. JNDI es un API que permite acceder de modo uniforme a múltiples servicios de directorio. Nos va a permitir localizar estos servicios a través de su nombre.

Para acceder al pool de conexiones debemos **definir el tipo de acceso a la base de datos que va a realizar la aplicación web como un recurso dentro del contexto de la aplicación.** Para ello deberemos abrir el fichero server.xml de nuestro servidor (También se puede crear un fichero context.xml dentro de la carpeta META-INF). Este fichero contiene una sección de contexto por cada aplicación web desplegada.

En nuestro caso la aplicación web se llama “AccesoBD”, dentro de la etiqueta de contexto añadimos un “Recurso” (Resource) que nos conecta a la base de datos.

```
<Context path="/AccesoBD" docBase = "AccesoBD" debug="5" reloadable="true" crossContext="true">
        <Resource name="jdbc/miDataSource" auth="Container" type="javax.sql.DataSource"
         driverClassName="com.mysql.cj.jdbc.Driver"
         maxActive="100" maxIdle="30" maxWait="10000"
         username="root" password="root"
         url= “jdbc:mysql://localhost/prueba"
         />
</Context>
```

Vamos a ver que significan los parámetros que hemos usado:

- ✓ **maxActive:** Indica el número máximo de conexiones que pueden estar abiertas al mismo tiempo.
- ✓ **maxIdle:** El número máximo de conexiones inactivas que permanecerán abiertas, si el número de conexiones inactivas es muy bajo puede darse el caso de que las conexiones se cierren porque se llega al máximo de conexiones inactivas y se vuelvan a abrir inmediatamente reduciendo la eficiencia ya que se perdería la ventaja del uso de pool de conexiones en cuanto a que no hay que abrir una conexión cada vez que es necesario.
- ✓ **maxWait:** Es el tiempo máximo (en ms) que se esperará a que haya una conexión disponible (inactiva), si se supera este tiempo se lanza una excepción.
- ✓ **username:** Usuario de la base de datos.
- ✓ **password:** Password para el usuario introducido en username.
- ✓ **driverClassName:** Nombre del driver JDBC para conectar con la base de datos.
- ✓ **url:** URL de la base de datos a la que nos queremos conectar.

Una vez dado de alta el pool de conexiones debemos modificar el fichero web.xml y registrarlo como un recurso accesible. En nuestro caso como trabajamos con la especificación Servlets 4.0 podemos utilizar la anotación **@Resource(name="jdbc/miDataSource").** Por último, modificamos el código del servlet para que utilice el pool de conexiones.

InitialContext se configura como una aplicación web que se implementa inicialmente y se pone a disposición de los componentes de la aplicación web (para el acceso de solo lectura). Todas las entradas y recursos configurados se colocarán en la parte **java:comp/env** del espacio de nombres JNDI, por lo que un acceso típico a un recurso, en este caso, a un JDBC DataSource, se implementaría de la siguiente forma:

```
// Obtain our environment naming context
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");

// Look up our data source
DataSource ds = (DataSource)
  envCtx.lookup("jdbc/EmployeeDB");

// Allocate and use a connection from the pool
Connection conn = ds.getConnection();
... use this connection to access the database ...
conn.close();
```

3.1.2.1 Ejemplo Pool de Conexiones

context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
 <Resource name="jdbc/miDataSource" auth="Container" type="javax.sql.DataSource"
           maxActive="50" maxIdle="30" maxWait="10000"
           username="root" password=""
           driverClassName="com.mysql.cj.jdbc.Driver"
           url="jdbc:mysql://localhost:3306/prueba"/>
</Context>
```

ServletPool.java

```
public class ServletPool extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public ServletPool() {
        super();
        // TODO Auto-generated constructor stub
    }


protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/plain");
    Connection conn=null;
    try {
        DataSource ds = (DataSource) new InitialContext().lookup("java:comp/env/jdbc/miDataSource");
        conn = ds.getConnection();
        if (conn != null)
            response.getWriter().append("Conexión creada!!");
        else
            response.getWriter().append("Conexión fallida!!");

    } catch (NamingException e) {
        // TODO Auto-generated catch block
        System.err.println("No se puede obtener el DataSource");
        e.printStackTrace();
    } catch (SQLException e) {
        System.err.println("Error al conectar a la BBDD");
        e.printStackTrace();
    } finally {
        try {
            conn.close();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```