Universidad Andrés Bello®

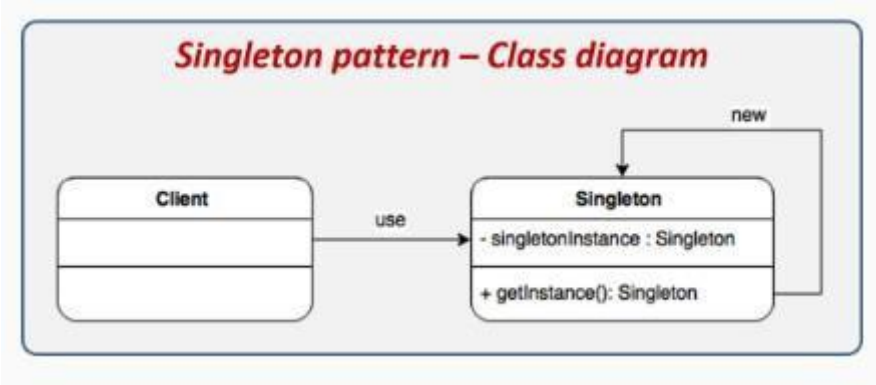
Conectar • Innovar • Liderar

DESARROLLO DE APLICACIONES WEB DINÁMICAS JAVA

El Patrón Singleton

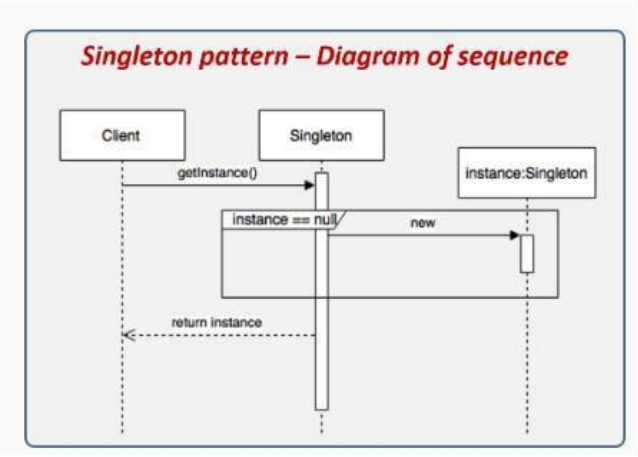
El patrón de diseño Singleton recibe su nombre debido a que sólo se puede tener una única instancia para toda la aplicación de una determinada clase, esto se logra restringiendo la libre creación de instancias de esta clase mediante el operador new e imponiendo un constructor privado y un método estático para poder obtener la instancia.

La intención de este patrón es garantizar que solamente pueda existir una única instancia de una determinada clase y que exista una referencia global en toda la aplicación.



- **Client:** Componente que desea obtener una instancia de la clase Singleton.
- **Singleton:** Clase que implementa el patrón Singleton, de la cual únicamente se podrá tener una instancia durante toda la vida de la aplicación.

Diagrama de Secuencia Patrón Singleton



1. El cliente solicita la instancia al Singleton mediante el método estático getInstance.
2. El Singleton validará si la instancia ya fue creada anteriormente, de no haber sido creada entonces se crea una nueva.
3. Se regresa la instancia creada en el paso anterior o se regresa la instancia existente en otro caso.

Ejemplo del mundo real

Mediante la implementación del patrón de diseño Singleton crearemos una aplicación que permite gestionar la configuración del sistema desde un único punto centralizado. Así, cuando la aplicación inicie, cargará la configuración inicial y está disponible para toda la aplicación.

Conexión a una Base de Datos mediante la biblioteca JDBC

¿Qué es JDBC?

JDBC es la especificación JavaSoft de una interfaz de programación de aplicaciones (API) estándar que permite que los programas Java accedan a sistemas de gestión de bases de datos. La API JDBC consiste en un conjunto de interfaces y clases escritas en el lenguaje de programación Java.

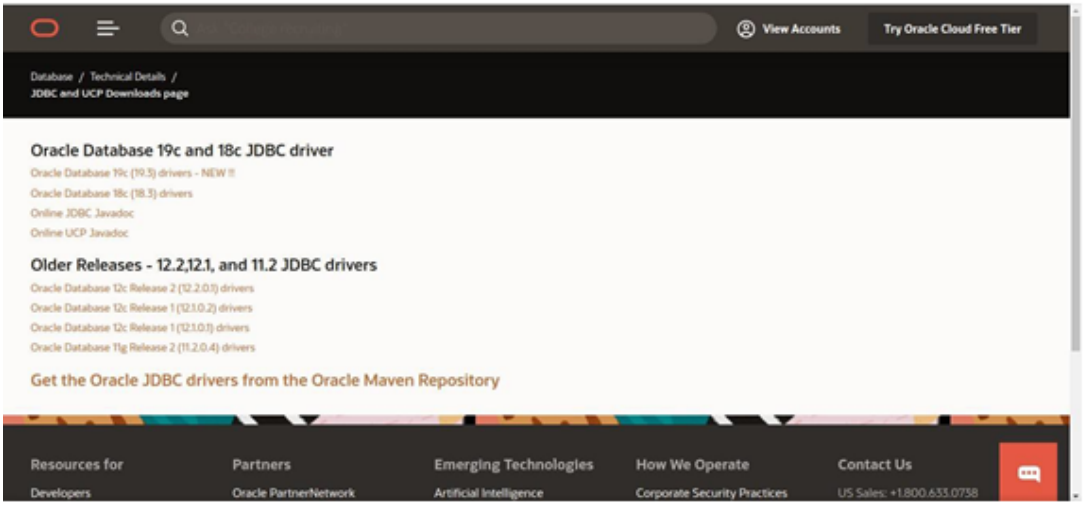
Con estas interfaces y clases estándar, los coders pueden escribir aplicaciones que conecten con bases de datos, envíen consultas escritas en el lenguaje de consulta estructurada (SQL) y procesan los resultados.

¿Qué hace JDBC?

- Establece una conexión con una BD
- Envía sentencias SQL
- Procesa los resultados

Descargar el controlador Oracle JDBC

Visite el sitio web de la base de datos Oracle ingresando al siguiente link https://www.oracle.com/database/technologies/appdev/jdbc-ucp-183-downloads.html y descargue el controlador Oracle JDBC; este controlador puede variar su distribución según el entorno de trabajo que se utilizará para crear la aplicación en Java.

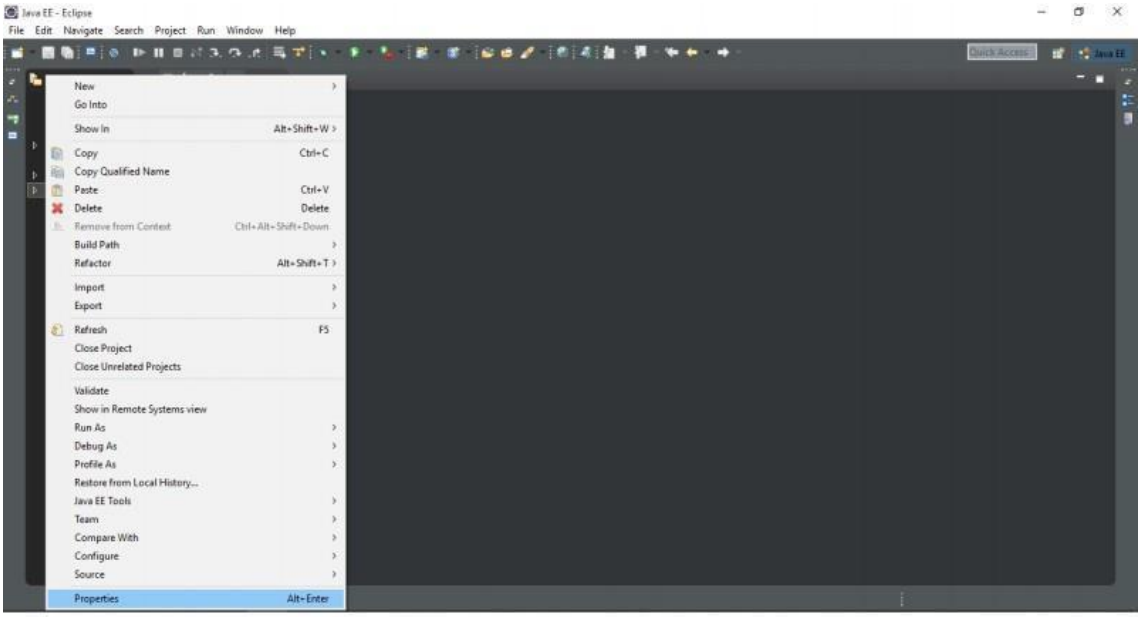


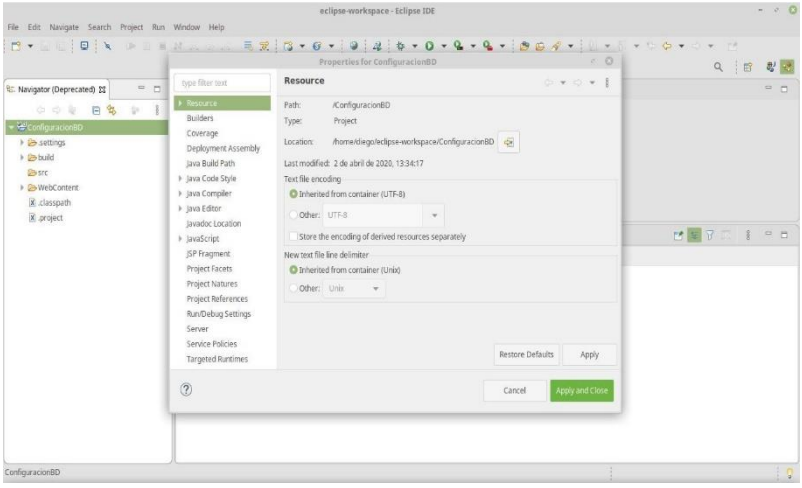
Añadir el conector a Eclipse

Una vez realizada la descarga, descomprimiremos el archivo en el directorio que deseemos. Hecho esto, iremos a Eclipse a la ventana donde se encuentra nuestro proyecto.

Desde las propiedades del proyecto de tu proyecto, pulsa sobre tu proyecto haciendo click en el botón derecho del mouse, y seleccionando la opción “properties”.

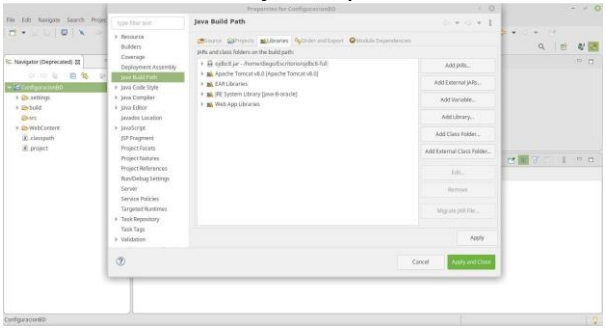
Se abrirá una pestaña como la siguiente:





Una vez aquí nos dirigiremos a la opción JAVA BUILD PATH, donde deberemos seleccionar la opción Add External JARs para añadir el archivo ".jar" de nuestro conector. En este caso para el ejemplo utilizaremos la versión JDBC8.

Una vez seleccionado nuestro archivo jar, se presiona el botón "Apply and Close".



Ejemplo 1: Cree una clase nombre de JDBCExample, que permita a una aplicación conectarse a una base de datos

```
import java.sql.*;
```

```
public class JDBCExample {
    public static void main(String[] args)
     {
          try {
               Connection conn = DriverManager.getConnection
               (
                    "jdbc:oracle:thin:@localhost:1521:xe",
                    "usuario",
                    "clave"
               )

            if (conn != null) {
                System.out.println("Conexión
                    correcta!");
            } else {
                System.out.println("Fallo
                    en la conexión!");
            }
        } catch (SQLException e) {
            System.err.format(
                    "SQL State: %s\n%s",
                    e.getSQLState(),
                    e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
           }
     }
}
```

Implementando un objeto singleton para la conexión a la base dedatos

Paso a Paso

Crearemos una nueva clase, por ejemplo EjemploSingleton, que será la responsable de crear la única instancia.

```
import java.sql.*;

public class EjemploSingleton {

}
```

Lo siguiente será crear una variable estática que almacenará nuestra conexión, y la inicializaremos a **null**.

```
private static Connection conn = null;
```

Después de eso deberemos crear un constructor para nuestra nueva clase, importante crearlo privado.

¿Por qué un constructor privado?

De esta manera vamos a garantizar que solo tendremos una única instancia para la conexión.

Ejemplo 2: usando JDBC Thin Driver

NOTA :
No requiere de la instalación cliente de ORACLE.
Ideal para Applets.

```
import java.sql.*;

class Conexion {

      public static void main
      (String args []) throws SQLException
      {
            DriverManager.registerDriver
            (new oracle.jdbc.driver.OracleDriver());

            Connection conn = DriverManager.getConnection
                        ("jdbc:oracle:thin:@localhost:1521:xe",
                                    "usuario",
                                    "clave");

            Statement stmt = conn.createStatement();
            ResultSet rset = stmt.executeQuery
            ("select BANNER from SYS.V_$VERSION");

            while (rset.next())
                  System.out.println
                  (rset.getString(1));
```

```
            stmt.close();
      }
}
```

El driver JDBC OCI permite realizar llamadas a ORACLE OCI directamente desde Java proporcionando un alto grado de compatibilidad con una versión específica de ORACLE utilizando métodos nativos, pero específicos de la plataforma.

Ejemplo 3: usando JDBC OCI - API nativa de ORACLE para aplicaciones J2EE

```
import java.sql.*;

class OracleCon{
     public static void main(String args[]){

          try{
               //paso 1: carga la clase del controlador
               Class.forName
               ("oracle.jdbc.driver.OracleDriver");

               //paso 2: crear la conexión con el objeto
               Connection con=DriverManager.getConnection(
               "jdbc:oracle:thin:@localhost:1521:xe",
               "usuario",
               "clave");

               //paso 3: crear objeto statement
               Statement stmt=con.createStatement();

               //paso 4: ejecutar query
               ResultSet rs = stmt.executeQuery
               ("select * from cliente");

               //paso 5: leer resultado registro a registro
               while(rs.next()) {
                    System.out.println(rs.getInt(1) +
                    "  " + rs.getString(2) +
                    "  " + rs.getString(3));
               }

               //paso 6: cierra el objeto de conexión
               con.close();
          }
          catch(Exception e){
               System.out.println(e);
```

Se entiende, pero si es privado, ¿cómo vamos a crearlo?. He ahí la gracia: al constructor lo llama la propia clase, a través de un método. Es lo que veremos a continuación.

Vamos a crear un método llamado getConnection en el que indicaremos que si nuestra propiedad conn es nula, vuelva a levantar el driver y crear una conexión llamando al constructor privado.

Y retornaremos la conexión para trabajar con ella.

```
public static Connection getConnection(){

    if (conn == null){
 new EjemploSingleton();
    }

    return conn;
}
```

Y una vez programada nuestra clase, podremos echar mano de esa conexión desde cualquier parte de nuestro programa llamándolo de esta forma.

```
Connection conn = EjemploSingleton.getConnection();
```

Ya tenemos un patrón de instancia única en Java con Oracle.

Ejemplo 4: Código completo clase

```
import java.sql.*;

public class EjemploSingleton {
```

```
    // Propiedades
    private static Connection conn = null;

    // Constructor
    private EjemploSingleton(){
          String url =
          "jdbc:oracle:thin:@localhost:1521:xe";

          String driver =
          "oracle.jdbc.driver.OracleDriver";

          String usuario = "usuario";
          String password = "clave";

          try{
               Class.forName(driver);
               conn = DriverManager.getConnection
               (url, usuario, password);
          }
          catch (ClassNotFoundException
                    | SQLException e){
               e.printStackTrace();
          }
     } // Fin constructor

     // Métodos
     public static Connection getConnection(){
          if (conn == null){
               new EjemploSingleton();
          }

          return conn;
     } // Fin getConnection
}
```

Para utilizar la conexión creada en el ejemplo anterior, se puede seguir la misma idea de los ejemplos anteriores. En este caso el procedimiento es bastante más simple, ya que no se generará una nueva conexión por cada consulta realizada.

Ejemplo 5: Utilización de conexión por medio de Singleton

```
import java.sql.*;

public class UsoSingleton {

      public static void main(String[] args) {

            try{
                  //Paso 1: Obtener una conexión
                  Connection con =
                              EjemploSingleton.getConnection();

                  //paso 2: crear objeto statement
                  Statement stmt=con.createStatement();

                  //paso 3: ejecutar query
                  ResultSet rs = stmt.executeQuery
                  ("select * from cliente");

                  //paso 4: leer resultado registro a registro
                  while(rs.next()) {
                        System.out.println(rs.getInt(1) +
                        "   " + rs.getString(2) +
                        "   " + rs.getString(3));
                  }

                  //paso 5: cierra el objeto de conexión
                  con.close();
            }
            catch(Exception e){
                  System.out.println(e);
            }
      }
}
```

La capa de acceso a datos (DAL)

Qué es la capa de acceso a datos

Una capa de acceso a datos (DAL) es una parte de un programa de computadora que proporciona acceso simplificado a los datos almacenados en algún tipo de almacenamiento persistente, como una base de datos relacional de entidades.

El acrónimo DAL se puede utilizar en varios contextos. Por ejemplo, el DAL podría devolver una referencia a un objeto (en términos de programación orientada a objetos) completo con sus atributos en lugar de una fila de campos de una tabla de base de datos. Esto permite que los módulos de cliente (o usuario) se creen con un mayor nivel de abstracción. Este tipo de modelo podría implementarse creando una clase de métodos de acceso a datos que hagan referencia directamente a un conjunto correspondiente de procedimientos almacenados de base de datos. Otra implementación podría potencialmente recuperar o escribir registros hacia o desde un sistema de archivos.

El DAL oculta esta complejidad del almacén de datos subyacente del mundo externo. Por ejemplo, en lugar de usar comandos como insertar, eliminar y actualizar para acceder a una tabla específica en una base de datos, se podrían crear una clase y algunos procedimientos almacenados en la base de datos. Los procedimientos se llamarían desde un método dentro de la clase, que devolvería un objeto que contiene los valores solicitados. O bien, los comandos de inserción, eliminación y actualización podrían ejecutarse dentro de funciones simples como registeruser o loginuser almacenadas dentro de la capa de acceso a datos.

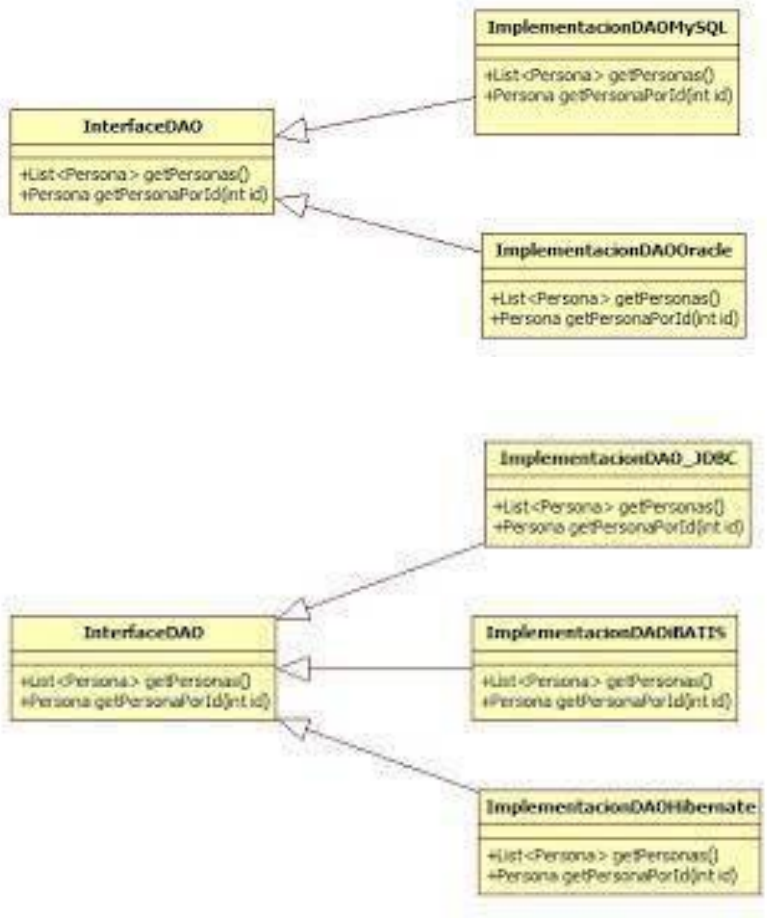
Además, los métodos de lógica empresarial de una aplicación se pueden asignar a la capa de acceso a datos. Entonces, por ejemplo, en lugar de realizar una consulta en una base de datos para buscar a todos los usuarios de varias tablas, la aplicación puede llamar a un solo método desde un DAL que abstrae esas llamadas a la base de datos.

Las aplicaciones que utilizan una capa de acceso a datos pueden depender del servidor de base de datos o ser independientes. Si la capa de acceso a datos admite varios tipos de bases de datos, la aplicación puede utilizar cualquier base de datos con la que pueda hablar el DAL. En cualquier circunstancia, tener una capa de acceso a datos proporciona una ubicación centralizada para todas las llamadas a la base de datos y, por lo tanto, hace que sea más fácil portar la aplicación a otros sistemas de base de datos (asumiendo que el 100% de la interacción de la base de datos se realiza en la DAL para un determinado solicitud).

El patrón DAO

DAO (en inglés, data access object, abreviado DAO) es un patrón de diseño utilizado para crear esta capa de persistencia. Tiene como finalidad separar la capa lógica del negocio del proyecto con la capa de persistencia.

DAO encapsula el acceso a la base de datos. Por lo que cuando la capa de lógica de negocio necesite interactuar con la base de datos, va a hacerlo a través de la API que le ofrece DAO. Generalmente esta API consiste en métodos CRUD (Create, Read, Update y Delete). Entonces por ejemplo cuando la capa de lógica de negocio necesite guardar un dato en la base de datos, va a llamar a un método create(). Lo que haga este método, es problema de DAO y depende de cómo DAO implemente el método create(), puede que lo implemente de manera que los datos se almacenen en una base de datos relacional como puede que lo implemente de manera que los datos se almacenen en ficheros de texto. Lo importante es que la capa de lógica de negocio no tiene porque saberlo, lo único que sabe es que el método create() va a guardar los datos, así como el método delete() va a eliminarlos, el método update() actualizarlos, etc. Pero no tiene idea de como interactúa DAO con la base de datos.



Implementando un DAO con métodos CRUD

Pasos generales para crear un CRUD en Java

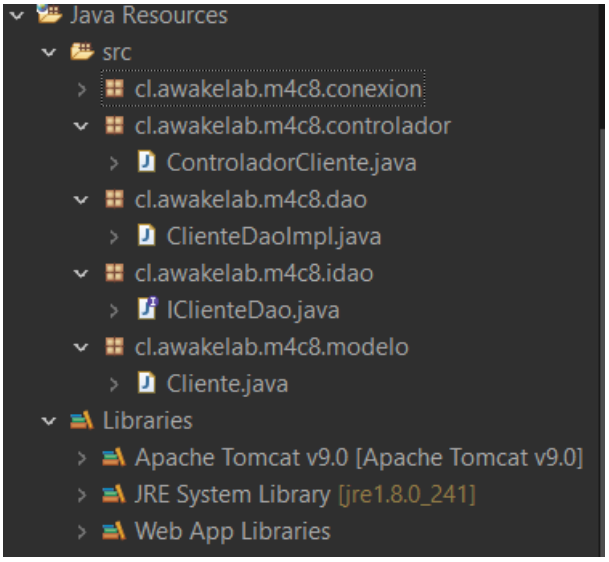
1. Crear una base de datos y una tabla.
2. Crear un nuevo proyecto Java.
3. Conectarte a la base de datos a través de JDBC Driver.
4. Programar los métodos para el CRUD.
5. Usar los métodos CRUD.

Crear la base de datos

Antes que nada debes crear una base de datos en Oracle llamada “ventas”, con una tabla llamada “cliente”, con los siguientes campos id: cedula, nombre y apellido; el campo id debe ser autoincrementable.

Crear un nuevo proyecto Java

Lo siguiente que vas a hacer es crear un proyecto en Java, ya que el CRUD lo vamos a hacer desde la consola; la estructura del proyecto debe quedar de la siguiente manera:



8.3.4.- Crear la conexión a traves de JDBC Driver

Una vez creado el proyecto, se debe crear la conexión de la aplicación a la base de datos; para esto se debe descargar el driver de conexión (JDBC) y agregarlo al class path del proyecto.

Programar los métodos CRUD

Ahora vamos a programar las operaciones CRUD utilizando el patrón de diseño DAO para la parte del acceso a los datos.

Primero empezamos creando el modelo que es la clase Cliente.java y que es la clase que va mapear los atributos de la tabla cliente en la base de datos “ventas”.

```
package cl.ejemplo.m4c8.modelo;

public class Cliente {

	private int id;
	private String cedula;
	private String nombre;
	private String apellido;

	public Cliente() {
		super();
	}

	public Cliente(int id,
			String cedula, String nombre,
			String apellido) {
		super();
		this.id = id;
		this.cedula = cedula;
		this.nombre = nombre;
		this.apellido = apellido;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}
	public String getCedula() {
		return cedula;
	}

	public void setCedula(String cedula) {
		this.cedula = cedula;
	}
```

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

	public String getApellido() {
		return apellido;
	}

	public void setApellido(String apellido) {
		this.apellido = apellido;
	}

	@Override
	public String toString() {
		return "Cliente [id="
				+ id + ", cedula="
				+ cedula + ", nombre="
				+ nombre + ", apellido="
				+ apellido + "]";
	}
}
```

Luego creamos la Interface IClienteDAO.java que registra los métodos CRUD, al ser una interface solo contiene la firma de los métodos asociados a la clase “ventas”.

```
package cl.ejemplo.m4c8.idao;

import java.util.List;

import cl.ejemplo.m4c8.modelo.Cliente;

public interface IClienteDao {

	public boolean registrar(Cliente cliente);
	public List<Cliente> obtener();
	public boolean actualizar(Cliente cliente);
	public boolean eliminar(Cliente cliente);

}
```

Lo siguiente es crear la implementación de la interface anterior, para esto lo hacemos en la clase ClienteDaoImpl.java, nota que también utilizamos la clase Conexion.java, para la ejecución de comandos SQL.

```
package cl.ejemplo.m4c8.dao;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

import cl.ejemplo.m4c8.conexion.EjemploConexion;
import cl.ejemplo.m4c8.idao.IClienteDao;
import cl.ejemplo.m4c8.modelo.Cliente;

public class ClienteDaoImpl implements IClienteDao {

	@Override
	public boolean registrar(Cliente cliente) {

		boolean registrar = false;

		Statement stm = null;
		Connection con = null;

		String sql = "INSERT INTO cliente values ('"
				+ cliente.getId() + "','"
				+ cliente.getCedula() + "','"
				+ cliente.getNombre() + "','"
				+ cliente.getApellido() + "')";

		try {
			con = EjemploConexion.conectar();
			stm = con.createStatement();
			stm.execute(sql);
			registrar = true;
			stm.close();
			con.close();
		}catch(SQLException e) {
			System.out.println("Error: "
					+ "Clase ClienteDaoImpl, "
					+ "método registrar");
			e.printStackTrace();
		}

		return registrar;
	}
```

```
@Override

public List<Cliente> obtener() {
      Connection con = null;
      Statement stm = null;
      ResultSet rs = null;

      String sql = "SELECT * FROM "
                  + "cliente ORDER BY id";
      List<Cliente> listaCliente
            = new ArrayList<Cliente>();

      try {
            con = EjemploConexion.conectar();
            stm = con.createStatement();
            rs = stm.executeQuery(sql);

            while (rs.next()) {
                  Cliente c = new Cliente();
                  c.setId(rs.getInt(1));
                  c.setCedula(rs.getString(2));
                  c.setNombre(rs.getString(3));
                  c.setApellido(rs.getString(4));
                  listaCliente.add(c);
            }

            rs.close();
            stm.close();
            con.close();
      }catch(SQLException e) {
            System.out.println("Error: "
                        + "Clase ClienteDaoImpl, "
                        + "método obtener");
            e.printStackTrace();
      }

      return listaCliente;
}

@Override
public boolean actualizar(Cliente cliente) {

      Connection con = null;
      Statement stm = null;

      boolean actualizar =  false;

      String sql = "UPDATE cliente SET "
                  + "cedula = '" + cliente.getCedula()
                  + "', "
```

```
                    + "nombre = '" + cliente.getNombre()
                                                "', "
                    + "apellido = '" + cliente.getApellido()
                    + "' "
                    + "WHERE id = '" + cliente.getId()
                    + "'";
            try {
                  con = EjemploConexion.conectar();
                  stm = con.createStatement();
                  stm.execute(sql);
                  actualizar = true;
                  stm.close();
                  con.close();
            }catch(SQLException e) {
                  System.out.println("Error: "
                              + "Clase ClienteDaoImpl, "
                              + "método actualizar");
                  e.printStackTrace();
            }

            return actualizar;
      }

      @Override
      public boolean eliminar(Cliente cliente) {
            Connection con = null;
            Statement stm = null;

            boolean eliminar = false;

            String sql = "DELETE FROM cliente "
                        + "WHERE id = '" + cliente.getId() + "'";

            try {
                  con = EjemploConexion.conectar();
                  stm = con.createStatement();
                  stm.execute(sql);
                  eliminar = true;
                  stm.close();
                  con.close();
            }catch(SQLException e) {
                  System.out.println("Error: "
                              + "Clase ClienteDaoImpl, "
                              + "método eliminar");
                  e.printStackTrace();
                              }

            return eliminar;
      }
}
```

Usar los métodos CRUD

El último paso del proceso sería utilizar los métodos correspondientes al CRUD de la clase Cliente. Para lograr esto, solo se necesita crear una instancia de la clase ClienteDaoImpl, y llamar a los métodos declarados en ella.

Lo anterior se puede hacer a través de un servlet, o bien desde una clase cualquiera. El resultado de cada proceso puede ser desplegado en un documento JSP a través de JSTL, o bien directamente en consola de comandos.

A continuación se indica un ejemplo de un servlet que hace un llamado a la clase de implementación, y que obtiene el listado de registros desde la tabla “cliente”. Este resultado se obtiene en forma de lista, y se envía a un JSP para su posterior despliegue.

```
package cl.awakelab.m4c8.controlador;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import cl.awakelab.m4c8.dao.ClienteDaoImpl;
import cl.awakelab.m4c8.modelo.Cliente;

@WebServlet("/ControladorCliente")
public class ControladorCliente extends HttpServlet {
     private static final long serialVersionUID = 1L;

    public ControladorCliente() {
        super();
    }

     protected void doGet(
                HttpServletRequest  request,
                HttpServletResponse response
                ) throws ServletException, IOException {
          ClienteDaoImpl clidao = new ClienteDaoImpl();
          List<Cliente> listaclientes
             =     new ArrayList<Cliente>();
```

```
            listaclientes = clidao.obtener();
            request.setAttribute("lclientes", listaclientes);
            request.getRequestDispatcher
            ("listaclientes.jsp").forward(request, response);
      }

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

El JSP que permite desplegar los datos se indica a continuación.

```
<%@ page language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<head>
      <meta charset="ISO-8859-1">
      <title>Módulo 4 - Día 8</title>
</head>
<body>
      <h1>Lista de clientes</h1>

      <table>
            <tr>
                  <th>ID</th>
                  <th>Cédula</th>
                  <th>Nombre</th>
                  <th>Apellido</th>
            </tr>
            <c:forEach items="${lclientes}" var="cli">
                  <tr>
                        <td>
                        <c:out value="${cli.getId()}" />
                        </td>
                        <td>
                        <c:out value="${cli.getCedula()}" />
                        </td>
                        <td>
```

```
                        <c:out value="${cli.getNombre()}" />
                        </td>
                        <td>
                        <c:out value="${cli.getApellido()}" />
                        </td>
                    </tr>
                </c:forEach>
            </table>
</body>
</html>
```

Anexo: Referencias

[1] Qué es Java Enterprise (J2ee, JEE)
Referencia: https://www.fundesem.es/bt/publicacion-java-ee-y-el-desarrollo-web-un-enfoque-de-aprendizaje

[2] Qué son los Servlets
Referencia: https://users.dcc.uchile.cl/~jbarrios/servlets/general.html

[3] Utilizando métodos post y get
Referencia: https://www.ecodeup.com/enviar-parametros-servlet-desde-una-vist a-utilizando-post-get/

[4] Sesiones y cookies
Referencia: https://ricardogeek.com/manejo-de-sesiones-y-cookies-en-servlets-java/

[5] Etiquetas JSTL
Referencia: http://www.jtech.ua.es/ayto/ayto2008/jsp/sesion07-apuntes.html

[6] MVC
Referencia: https://www.ecodeup.com/patrones-de-diseno-en-java-mvc-dao-y-dto/

[7] ¿Qué son los patrones de diseño de software?
Referencia: https://profile.es/blog/patrones-de-diseno-de-software/

[8] Patrón Singleton
Referencias: https://reactiveprogramming.io/blog/es/patrones-de-diseno/singleton