How do I implement a DAO manager using JDBC and connection pools?

up vote16down vote**31**
My problem is as follows. I need a class that works as a single point to a database connection in a web system, so to avoid having one user with two open connections. I need it to be as optimal as possible and it should manage every transaction in the system. In other words only that class should be able to instantiate DAOs. And to make it better, it should also use connection pooling! What should I do?

up vote56down voteaccepted
You will need to implement a **DAO Manager**. I took the main idea from this website, however I made my own implementation that solves some few issues.

Step 1: Connection pooling

First of all, you will have to configure a **connection pool**. A connection pool is, well, a pool of connections. When your application runs, the connection pool will start a certain amount of connections, this is done to avoid creating connections in runtime since it's a expensive operation. This guide is not meant to explain how to configure one, so go look around about that.

For the record, I'll use **Java** as my language and **Glassfish** as my server.

Step 2: Connect to the database

Let's start by creating a `DAOManager` class. Let's give it methods to open and close a connection in runtime. Nothing too fancy.

```
public class DAOManager {

    public DAOManager() throws Exception {
        try
        {
            InitialContext ctx = new InitialContext();
            this.src = (DataSource)ctx.lookup("jndi/MYSQL"); //The string should be the
same name you're giving to your JNDI in Glassfish.
        }
        catch(Exception e) { throw e; }
    }

    public void open() throws SQLException {
        try
        {
            if(this.con==null || !this.con.isOpen())
                this.con = src.getConnection();
        }
        catch(SQLException e) { throw e; }
    }

    public void close() throws SQLException {
        try
        {
            if(this.con!=null && this.con.isOpen())
                this.con.close();
        }
        catch(SQLException e) { throw e; }
    }

    //Private
    private DataSource src;
    private Connection con;

}
```

This isn't a very fancy class, but it'll be the basis of what we're going to do. So, doing this:

```
DAOManager mngr = new DAOManager();
mngr.open();
mngr.close();
```

should open and close your connection to the database in an object.

Step 3: Make it a single point!

What, now, if we did this?

```
DAOManager mngr1 = new DAOManager();
DAOManager mngr2 = new DAOManager();
mngr1.open();
mngr2.open();
```

Some might argue, *"why in the world would you do this?"*. But then you never know what a programmer will do. Even then, the programmer might forger from closing a connection before opening a new one. Plus, this is a waste of resources for the application. **Stop here if you actually want to have two or more open connections, this will be an implementation for one connection per user.**

In order to make it a single point, we will have to convert this class into a **singleton**. A singleton is a design pattern that allows us to have one and only one instance of any given object. So, let's make it a singleton!

- We must convert our `public` constructor into a private one. We must only give an instance to whoever calls it. The `DAOManager` then becomes a factory!
- We must also add a new `private` class that will actually store a singleton.
- Alongside all of this, we also need a `getInstance()` method that will give us a singleton instance we can call.

Let's see how it's implemented.

```
public class DAOManager {

    public static DAOManager getInstance() {
        return DAOManagerSingleton.INSTANCE;
    }

    public void open() throws SQLException {
        try
        {
            if(this.con==null || !this.con.isOpen())
                this.con = src.getConnection();
        }
        catch(SQLException e) { throw e; }
    }

    public void close() throws SQLException {
        try
        {
            if(this.con!=null && this.con.isOpen())
                this.con.close();
        }
        catch(SQLException e) { throw e; }
    }

    //Private
    private DataSource src;
    private Connection con;

    private DAOManager() throws Exception {
        try
        {
            InitialContext ctx = new InitialContext();
            this.src = (DataSource)ctx.lookup("jndi/MYSQL");
        }
        catch(Exception e) { throw e; }
    }

    private static class DAOManagerSingleton {

        public static final DAOManager INSTANCE;
        static
        {
            DAOManager dm;
            try
            {
                dm = new DAOManager();
            }
            catch(Exception e)
                dm = null;
            INSTANCE = dm;
        }

    }

}
```

When the application starts, whenever anyone needs a singleton the system will instantiate one `DAOManager`. Quite neat, we've created a single access point!

But singleton is an antipattern because reasons! I know some people won't like singleton. However it solves the problem (and has solved mine) quite decently. This is just a way of implementing this solution, if you have other ways you're welcome to suggest so.

Step 4: But there's something wrong...

Yes, indeed there is. **A singleton will create only ONE instance for the whole application!** And this is wrong in many levels, especially if we have a web system where our application will be **multithreaded**! How do we solve this, then?

Java provides a class named `ThreadLocal`. A `ThreadLocal` variable will have one instance per thread. Hey, it solves our problem! See more about how it works, you will need to understand its purpose so we can continue.

Let's make our `INSTANCE` `ThreadLocal` then. Modify the class this way:

```
public class DAOManager {

    public static DAOManager getInstance() {
        return DAOManagerSingleton.INSTANCE.get();
    }

    public void open() throws SQLException {
        try
        {
            if(this.con==null || !this.con.isOpen())
                this.con = src.getConnection();
        }
        catch(SQLException e) { throw e; }
    }

    public void close() throws SQLException {
        try
        {
            if(this.con!=null && this.con.isOpen())
                this.con.close();
        }
        catch(SQLException e) { throw e; }
    }

    //Private
    private DataSource src;
    private Connection con;

    private DAOManager() throws Exception {
        try
        {
            InitialContext ctx = new InitialContext();
            this.src = (DataSource)ctx.lookup("jndi/MYSQL");
        }
        catch(Exception e) { throw e; }
    }

    private static class DAOManagerSingleton {

        public static final ThreadLocal<DAOManager> INSTANCE;
        static
        {
            ThreadLocal<DAOManager> dm;
            try
            {
                dm = new ThreadLocal<DAOManager>(){
                    @Override
                    protected DAOManager initialValue() {
                        try
                        {
                            return new DAOManager();
                        }
                        catch(Exception e)
                        {
                            return null;
                        }
                    }
                };
            }
            catch(Exception e)
                dm = null;
            INSTANCE = dm;
        }

    }

}
```

I would seriously love to not do this

```
catch(Exception e)
{
    return null;
}
```

but `initialValue()` can't throw an exception. Oh, `initialValue()` you mean? This method will tell us what value will the `ThreadLocal` variable hold. Basically we're initializing it. So, thanks to this we can now have one instance per thread.

Step 5: Create a DAO

A `DAOManager` is nothing without a DAO. So we should at least create a couple of them.

> A DAO, short for "Data Access Object" is a design pattern that gives the responsability of managing database operations to a class representing a certain table.

In order to use our `DAOManager` more efficiently, we will define a `GenericDAO`, which is an abstract DAO that will hold the common operations between all DAOs.

```
public abstract class GenericDAO<T> {
    public abstract int count() throws SQLException;

    //Protected
    protected final String tableName;
    protected Connection con;

    protected GenericDAO(Connection con, String tableName) {
        this.tableName = tableName;
        this.con = con;
    }

}
```

For now, that will be enough. Let's create some DAOs. Let's suppose we have two POJOs: `First` and `Second`, both with just a `String` field named `data` and its getters and setters.

```
public class FirstDAO extends GenericDAO<First> {

    public FirstDAO(Connection con) {
        super(con, TABLENAME);
    }

    @Override
    public int count() throws SQLException {
        String query = "SELECT COUNT(*) AS count FROM "+this.tableName;
        PreparedStatement counter;
        try
        {
        counter = this.con.PrepareStatement(query);
        ResultSet res = counter.executeQuery();
        res.next();
        return res.getInt("count");
        }
        catch(SQLException e){ throw e; }
    }

   //Private
   private final static String TABLENAME = "FIRST";

}
```

`SecondDAO` will have more or less the same structure, just changing `TABLENAME` to `"SECOND"`.

Step 6: Making the manager a factory

`DAOManager` not only should serve the purpose of serving as a single connection point. Actually, `DAOManager` should answer this question:

> Who is the one responsible of managing the connections to the database?

The individual DAOs shouldn't manage them, but `DAOManager`. We've answered partially the question, but now we shouldn't let anyone manage other connections to the database, not even the DAOs. But, the DAOs need a connection to the database! Who should provide it? `DAOManager` indeed! What we should do is making a factory method inside `DAOManager`. Not just that, but `DAOManager` will also hand them the current connection!

Factory is a design pattern that will allow us to create instances of a certain superclass, without knowing exactly what child class will be returned.

First, let's create an `enum` listing our tables.

```
public enum Table { FIRST, SECOND }
```

And now, the factory method inside `DAOManager`:

```
public GenericDAO getDAO(Table t) throws SQLException
{

    try
    {
        if(this.con == null || this.con.isClosed()) //Let's ensure our connection is open
            this.open();
    }
    catch(SQLException e){ throw e; }

    switch(t)
    {
    case FIRST:
        return new FirstDAO(this.con);
    case SECOND:
        return new SecondDAO(this.con);
    default:
        throw new SQLException("Trying to link to an unexistant table.");
    }

}
```

Step 7: Putting everything together

We're good to go now. Try the following code:

```
DAOManager dao = DAOManager.getInstance();
FirstDAO fDao = (FirstDAO)dao.getDAO(Table.FIRST);
SecondDAO sDao = (SecondDAO)dao.getDAO(Table.SECOND);
System.out.println(fDao.count());
System.out.println(sDao.count());
dao.close();
```

Isn't it fancy and easy to read? Not just that, but when you call `close()`, you close *every single connection* the DAOs are using. *But how?!* Well, they're sharing the same connection, so it's just natural.

Step 8: Fine-tuning our class

We can do several things from here on. To ensure connections are closed and returned to the pool, do the following in `DAOManager`:

```
@Override
protected void finalize()
{

    try{ this.close(); }
    finally{ super.finalize(); }

}
```

You can also implement methods that encapsulate `setAutoCommit()`, `commit()` and `rollback()` from the `Connection` so you can have a better handling of your transactions. What I also did is, instead of just holding a `Connection`, `DAOManager` also holds a `PreparedStatement` and a `ResultSet`. So, when calling `close()` it also closes both. A fast way of closing statements and result sets!

I hope this guide can be of any use to you in your next project!

up vote4down vote
I think that if you want to do a simple DAO pattern in plain JDBC you should keep it simple:

```
      public List<Customer> listCustomers() {
            List<Customer> list = new ArrayList<>();
            try (Connection conn = getConnection();
                 Statement s = conn.createStatement();
                 ResultSet rs = s.executeQuery("select * from customers")) {
```

```
                while (rs.next()) {
                    list.add(processRow(rs));
                }
                return list;
            } catch (SQLException e) {
                throw new RuntimeException(e.getMessage(), e); //or your exceptions
            }
        }
```

You can follow this pattern in a class called for example CustomersDao or CustomerManager, and you can call it with a simple

```
CustomersDao dao = new CustomersDao();
List<Customers> customers = dao.listCustomers();
```

Note that I'm using try with resources and this code is safe to connections leaks, clean, and straightforward, You probably don't want to follow the full DAO pattern with Factorys, interfaces and all that plumbing that in many cases don't add real value.

I don't think that it's a good idea using ThreadLocals, Bad used like in the accepted answer is a source of classloader leaks

Remember ALWAYS close your resources (Statements, ResultSets, Connections) in a try finally block or using try with resources