|  |  |
| --- | --- |
| **JDBC DataSource Example** | |
|  |  |
|  | Let’s create a simple JDBC project and learn how to use MySQL and Oracle DataSource basic implementation classes to get the database connection. |
|  | Our final project will look like below image. |
|  |  |
|  | 1.png |
|  |  |
|  | Database Setup |
|  | Before we get into our example programs, we need some database setup with table and sample data. (Installation of MySQL and ORACLE database must be done.) |
|  |  |
|  | 2.png |
|  | 3.png |
|  | Now let’s move on to our java programs. For having database configuration loosely coupled, I will read them from property file. |
|  | 4.png |
|  | Make sure that above configurations match with your local setup. Also make sure you have MySQL and Oracle DB JDBC jars included in the build path of the project. |
|  |  |
|  | JDBC MySQL and Oracle DataSource Example |
|  | Let’s write a factory class that we can use to get MySQL or Oracle DataSource. |
|  |  |
|  | 12.png |
|  | 6.png |
|  | 7.png |
|  | Notice that both Oracle and MySQL DataSource implementation classes are very similar, |
|  | let’s write a simple test program to use these methods and run some test. |
|  | 13.png |
|  | 9.png |
|  | System.out.println("Employee ID="+rs.getInt("empid")+ |
|  | ", Name="+rs.getString("name")); |
|  | 10.png |
|  |  |
|  | Notice that the client class is totally independent of any Database specific classes. This helps us in hiding the underlying implementation details from client program and achieve loose coupling and abstraction benefits. |
|  | When we run above test program, we will get below output. |
|  | 11.png |
|  |  |
|  | If you look at above DataSource factory class, there are two major issues with it. |
| 1. | The factory class methods to create the MySQL and Oracle DataSource are tightly coupled with respective driver API. If we want to remove support for Oracle database in future or want to add some other database support, it will require code change. |
| 2. | Most of the code to get the MySQL and Oracle DataSource is similar, the only different is the implementation class that we are using. |
|  |  |
|  | Apache Commons DBCP API helps us in getting rid of these issues by providing DataSource implementation that works as an abstraction layer between our program and different JDBC drivers. |
|  | Apache DBCP library depends on Commons Pool library, so make sure they both are in the build path as shown in the image. |
|  | Here is the DataSource factory class using BasicDataSource that is the simple implementation of DataSource. |
|  |  |
|  | 14.png |
|  | 15.png |
|  | 16.png |
|  | As you can see that depending on user input, either MySQL or Oracle datasource is created. If you are supporting only one database in the application then you don’t even need these logic. Just change the properties and you can switch from one database server to another. The key point through which Apache DBCP provide abstraction is setDriverClassName() method. |
|  | Here is the client program using above factory method to get different types of connection. |
|  |  |
|  | 17.png |
|  | 18.png |
|  | System.out.println("Employee ID="+rs.getInt("empid")+", Name="+rs.getString("name")); |
|  | 19.png |
|  |  |
|  | When you run above program, the output will be same as earlier program. |
|  | If you look at the DataSource and above usage, it can be done with normal DriverManager too. The major benefit of DataSource is when it’s used within a Context and with JNDI. |
|  | With simple configurations we can create a Database Connection Pool that is maintained by the Container itself. Most of the servlet containers such as Tomcat and JBoss provide it’s own DataSource implementation and all we need is to configure it through simple XML based configurations and then use JNDI context lookup to get the DataSource and work with it. |