# [Difference between RowCallbackHandler and RowMapper](#diff_RowCallbackHandler_RowMapper)

1. How to get Spring JDBC connection?

org.springframework.jdbc.core.

In context.xml under resources folder

<bean id=*"destDataSource"*

class=*"org.springframework.jdbc.datasource.SimpleDriverDataSource"*>

<property name=*"driverClass"* value=*"${dest.jdbc.driver}"* />

<property name=*"url"* value=*"${dest.jdbc.url}"* />

<property name=*"username"* value=*"${dest.jdbc.username}"* />

<property name=*"password"* value=*"${dest.jdbc.password}"* />

</bean>

to have transaction manager use

<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"destDataSource"* />

</bean>

For @trasaction annotation to be scanned add the following in context.xml file.

<tx:annotation-driven transaction-manager=*"transactionManager"* />

Use data source in java

**private** jaavax.sql.DataSource destDataSource = **null**;

**public** DataSource getDestDataSource() {

**return** destDataSource;

}

//@Resource(name = "destDataSource")

**public** **void** setDestDataSource(DataSource destDataSource) {

**this**.destDataSource = destDataSource;

**if**(destDataSource != **null**) {

**this**.*destJdbcTemplate* = **new** JdbcTemplate(**this**.destDataSource);

}

}

**In DAO**

**JdbcTemplate jdbcTemplate = getDestJdbcTemplate();**

**dtos = jdbcTemplate.query(getReviewSurvCycleListQuery, args, new ReviewSurvCycleMapper());**

**private static class ReviewSurvCycleMapper implements RowMapper<Map<String,SurvCycleDTO>>{ mapRow(ResultSet rs, int rowNum) }**

**RowMapper:** An interface used by [JdbcTemplate](eclipse-javadoc:%E2%98%82=Survillence_Data_Migration/C:%5C/Users%5C/cherukun%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-jdbc%5C/3.0.5.RELEASE%5C/spring-jdbc-3.0.5.RELEASE.jar%3Corg.springframework.jdbc.core(RowMapper.class%E2%98%83RowMapper%E2%98%82JdbcTemplate) for mapping rows of a [java.sql.ResultSet](eclipse-javadoc:%E2%98%82=Survillence_Data_Migration/C:%5C/Users%5C/cherukun%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-jdbc%5C/3.0.5.RELEASE%5C/spring-jdbc-3.0.5.RELEASE.jar%3Corg.springframework.jdbc.core(RowMapper.class%E2%98%83RowMapper%E2%98%82java.sql.ResultSet) on a per-row basis. Implementations of this interface perform the actual work of mapping each row to a result object, but don't need to worry about exception handling. [SQLExceptions](eclipse-javadoc:%E2%98%82=Survillence_Data_Migration/C:%5C/Users%5C/cherukun%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-jdbc%5C/3.0.5.RELEASE%5C/spring-jdbc-3.0.5.RELEASE.jar%3Corg.springframework.jdbc.core(RowMapper.class%E2%98%83RowMapper%E2%98%82java.sql.SQLException) will be caught and handled by the calling JdbcTemplate.

**In hibernate we use**

org.springframework.orm.hibernate3.HibernateTemplate

HibernateTemplate(sessionFactory);

1. Use map in the context.xml and use in java

<bean id=*"loadCorpUserInfoDaoImpl"* class=*"com.moodys.srv.datamigration.dao.LoadCorpUserInfoDaoImpl"*>

<property name=*"connData"*>

<map>

<entry key=*"java.naming.factory.initial"* value=*"com.sun.jndi.ldap.LdapCtxFactory"*/>

<entry key=*"java.naming.security.authentication"* value=*"simple"*/>

<entry key=*"java.naming.security.principal"* value=*"sys\_dev\_pfgmon@ad.moodys.net"*/>

<entry key=*"java.naming.provider.url"* value=*"ldap://ad.moodys.net"*/>

<entry key=*"java.naming.security.credentials"* value=*"Qtg5kh3kKaRzB1"*/>

</map>

</property>

</bean>

1. To Process rows from the query

jt.query(getCurrentStatusTypeCodeQuery, **new** RowCallbackHandler (){

**public** **void** processRow(ResultSet rs) **throws** SQLException {

result.put(JdbcRSUtils.*getStringNotNull*(rs, 1),

JdbcRSUtils.*getStringNotNull*(rs, 2));

}

});

|  |
| --- |
| Pom.xml    <dependency> |

|  |  |
| --- | --- |
| 25 | <groupId>org.springframework</groupId> |

|  |  |
| --- | --- |
| 26 | <artifactId>spring-jdbc</artifactId> |

|  |  |
| --- | --- |
| 27 | <version>${spring.version}</version> |

|  |  |
| --- | --- |
| 28 | </dependency> |

## [HOW TO GET AUTO-GENERATED KEY WITH JDBCTEMPLATE](http://www.javacreed.com/how-to-get-auto-generated-key-with-jdbctemplate/)

Spring provides GeneratedKeyHolder ([Java Doc](http://docs.spring.io/spring/docs/3.2.8.RELEASE/javadoc-api/org/springframework/jdbc/support/GeneratedKeyHolder.html)) class which can be used to retrieve the auto generated values.

All code listed below is available at: [https://java-creed-examples.googlecode.com/svn/spring/How to get Auto-Generated Key with JdbcTemplate/](https://java-creed-examples.googlecode.com/svn/spring/How%20to%20get%20Auto-Generated%20Key%20with%20JdbcTemplate/). Most of the examples will not contain the whole code and may omit fragments which are not relevant to the example being discussed. The readers can download or view all code from the above link.

The following class shows how to retrieve the auto generated key after a new value is added to the table.

package com.javacreed.examples.spring;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import java.sql.Statement;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.PreparedStatementCreator;

import org.springframework.jdbc.support.GeneratedKeyHolder;

import org.springframework.jdbc.support.KeyHolder;

public class ExampleDao {

@Autowired

private JdbcTemplate jdbcTemplate;

public long addNew(final String name) {

final PreparedStatementCreator psc = new PreparedStatementCreator() {

@Override

public PreparedStatement createPreparedStatement(final Connection connection) throws SQLException {

final PreparedStatement ps = connection.prepareStatement("INSERT INTO `names` (`name`) VALUES (?)",

Statement.RETURN\_GENERATED\_KEYS);

ps.setString(1, name);

return ps;

}

};

// The newly generated key will be saved in this object

**final KeyHolder holder = new GeneratedKeyHolder();**

jdbcTemplate.update(psc, **holder**);

**final long newNameId = holder.getKey().longValue();**

**return newNameId;**

}

}

**How to call Stored Procedures from Java using IN and OUT parameter example**

Spring Framework provides excellent support to call stored procedures from Java application. In fact there are multiple ways to call stored procedure in Spring Framework,

e.g. you can use one of the query() method from JdbcTemplate to call stored procedures,

or you can extend [abstract class](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html) StoredProcedure to call stored procedures from Java.

In this Java Spring tutorial, we will see second approach to call stored procedure. It's more [object oriented](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html), but same time requires more coding. StoredProcedure class allows you to declare IN and OUT parameters and call stored procedure using its various execute() method, which has protected access and can only be called from sub class. I personally prefer to implement StoredProcedure class as [Inner class](http://javarevisited.blogspot.sg/2012/12/inner-class-and-nested-static-class-in-java-difference.html), if its tied up with one of [DAO Object](http://javarevisited.blogspot.com/2013/01/data-access-object-dao-design-pattern-java-tutorial-example.html), e.g. in this case it nicely fit inside EmployeeDAO. Then you can provide convenient method to wrap stored procedure calls. In order to demonstrate, how to call stored procedures from spring based application, we will first create a simple stored proc using MySQL database, as shown below.  
  
Read more: <http://javarevisited.blogspot.com/2013/04/spring-framework-tutorial-call-stored-procedures-from-java.html#ixzz3AUAesShI>



Spring Stored Procedure example and Configurations

[**http://javarevisited.blogspot.com/2013/04/spring-framework-tutorial-call-stored-procedures-from-java.html**](http://javarevisited.blogspot.com/2013/04/spring-framework-tutorial-call-stored-procedures-from-java.html)

Your stored procedure can have multiple IN and OUT parameter. StoredProcedure class also provide several execute() methods, which can be invoked to call stored procedure and get result. It return result as [Map](http://javarevisited.blogspot.com/2011/12/how-to-traverse-or-loop-hashmap-in-java.html), where key is OUT parameter, and value is result of stored procedure. Here is the code for DAO class and stored procedure along with Spring Configuration file, since Spring framework is based on principle of [dependency Injection and Inversion of control](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html), this file is required to create and manage object.

**Java Class which wraps Stored procedure**

import java.sql.Types;

import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.SqlOutParameter;

import org.springframework.jdbc.core.SqlParameter;

import org.springframework.jdbc.object.StoredProcedure;

public class EmployeeDao {

private JdbcTemplate jdbcTemplate;

private EmployeeSP sproc;

public void setDataSource(DataSource source){

this.jdbcTemplate = new JdbcTemplate(source);

this.sproc = new EmployeeSP(jdbcTemplate.getDataSource());

}

*/\**

*\* wraps stored procedure call*

*\*/*

public String getEmployeeName(int emp\_id){

return (String) sproc.execute(emp\_id);

}

*/\**

*\* Inner class to implement stored procedure in spring.*

*\*/*

private class EmployeeSP extends *StoredProcedure*{

private static final String SPROC\_NAME = "usp\_GetEmployeeName";

public EmployeeSP( DataSource datasource ){

super( datasource, SPROC\_NAME );

declareParameter( new SqlParameter( "id", Types.INTEGER) ); *//declaring sql in parameter to pass input*

declareParameter( new SqlOutParameter( "name", Types.VARCHAR ) ); *//declaring sql out parameter*

compile();

}

public Object execute(int emp\_id){

Map<String,Object> results = super.execute(emp\_id);

return results.get("name"); *//reading output of stored procedure using out parameters*

}

}

}

**Main class to test stored procedure**

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

*/\**

*\* Main class to start and test this Java application*

*\*/*

public class Main {

public static void main(String args[]){

ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config.xml");

EmployeeDao dao = (EmployeeDao) ctx.getBean("employeeDao");

*//calling stored procedure using DAO method*

System.out.println("Employee name for id 103 is : " + dao.getEmployeeName(103));

}

}

Output

2013-01-17 23:56:34,408 0 [main] DEBUG EmployeeDao$EmployeeSP - Compiled stored procedure. Call string is [{call usp\_GetEmployeeName(?, ?)}]

2013-01-17 23:56:34,439 31 [main] DEBUG EmployeeDao$EmployeeSP - RdbmsOperation with SQL [usp\_GetEmployeeName] compiled

Employee name for id 103 is : Jack

**Spring configuration file:**

<?**xml** version="1.0" encoding="UTF-8"?>

<beans xmlns="http:**//**www.springframework.org/schema/beans"

        xmlns:xsi="http:**//**www.w3.org/2001/XMLSchema-instance" xmlns:p="http:**//**www.springframework.org/schema/p"

        xmlns:jms="http:**//**www.springframework.org/schema/jms" xmlns:context="http:**//**www.springframework.org/schema/context"

        xsi:schemaLocation="

http:**//**www.springframework.org/schema/beans http:**//**www.springframework.org/schema/beans/spring-beans-3.0.xsd

http:**//**www.springframework.org/schema/jms http:**//**www.springframework.org/schema/jms/spring-jms-3.0.xsd

http:**//**www.springframework.org/schema/context http:**//**www.springframework.org/schema/context/spring-context-3.0.xsd">

        <bean id="propertyPlaceholderConfigurer"

                class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

                <property name="locations">

                        <list>

                                <value>classpath:jdbc.properties</value>

                        </list>

                </property>

        </bean>

    <bean id="springDataSource" class="org.springframework.jdbc.datasource.SingleConnectionDataSource">

        <property name="driverClassName" value="${db.driver}" />

        <property name="url" value="${db.url}" />

        <property name="username" value="root" />

        <property name="password" value="root" />

    </bean>

        <bean id="employeeDao" class ="EmployeeDao">

                <property name="dataSource" ref="springDataSource"/>

        </bean>

</beans>

That's all on **How to call stored procedure from Java application using Spring Framework**. As discussed in 10 JDBC best practices for Java Programmer, JDBC API provides more straightforward approach using CallableStatement, but Spring's StoredProcedure class is also easy to use. You can also explore calling stored procedure, directly using JdbcTemplate in Spring.

Read more: <http://javarevisited.blogspot.com/2013/04/spring-framework-tutorial-call-stored-procedures-from-java.html#ixzz3AUBc6Mfg>

# Difference between RowCallbackHandler and RowMapper

<http://www.coderanch.com/t/447966/Spring/Difference-RowCallbackHandler-RowMapper>

Both RowCallBackHandler and RowMapper are both for retrieving the results from a [ResultSet](http://docs.oracle.com/javaee/7/api/javax/resource/cci/ResultSet.html" \t "_new" \o "Java API) which takes the responsibility to process each row of data in the [ResultSet](http://docs.oracle.com/javaee/7/api/javax/resource/cci/ResultSet.html" \t "_new" \o "Java API). Both interface methods should not call next() on the [ResultSet](http://docs.oracle.com/javaee/7/api/javax/resource/cci/ResultSet.html" \t "_new" \o "Java API). It requires to extract values of the current row and process the data on per-row basis.   
  
**RowCallBackHandler's processRow**([ResultSet](http://docs.oracle.com/javaee/7/api/javax/resource/cci/ResultSet.html" \t "_new" \o "Java API) rs) does not return object which is having the result. If require it can store the result in its instance variable and then make it available it to DAO but for this it requires to be designed as stateful.   
**So this type of method(processRow()) is generally implemented to process the result like preparing XML document with the result data.**   
  
But **RowMapper's mapRow([ResultSet](http://docs.oracle.com/javaee/7/api/javax/resource/cci/ResultSet.html" \t "_new" \o "Java API) rs, int rowNum)** can return an Object representing the result object for the current row.   
In general this interface is designed as stateless to make it reusable and reduce the memory requirements.

Examples:



Date in Sybase and java mapping

In database use date column as datetime and in java timestamp

| **Table 6-1: Simply and object mappable SQL and Java datatypes** | | |
| --- | --- | --- |
| **SQL datatype** | **Corresponding Java datatypes** | |
| **Simply mappable** | **Object mappable** |
| char/unichar |  | java.lang.*String* |
| nchar |  | java.lang.*String* |
| varchar/univarchar |  | java.lang.*String* |
| nvarchar |  | java.lang.*String* |
| text |  | java.lang.*String* |
| numeric |  | java.math.BigDecimal |
| decimal |  | java.math.BigDecimal |
| money |  | java.math.BigDecimal |
| smallmoney |  | java.math.BigDecimal |
| bit | boolean | Boolean |
| tinyint | byte | Integer |
| smallint | short | Integer |
| integer | int | Integer |
| bigint | long | java.math.BigInteger |
| unsigned smallint | int | Integer |
| unsigned int | long | Integer |
| unsigned bigint |  | java.math.BigInteger |
| real | float | Float |
| float | double | Double |
| double precision | double | Double |
| binary |  | byte[] |
| varbinary |  | byte[] |
| datetime |  | java.sql.Timestamp |
| smalldatetime |  | java.sql.Timestamp |
| date |  | java.sql.Date |
| time |  | java.sql.Time |

**What is difference between java.sql.Time and java.sql.TimeStamp in Java.**

Answer : This JDBC questions is similar to earlier JDBC interview question java.sql.Date vs java.util.Date. Main difference is that java.sql.Time class doesn't contain any date information on it while java.sql.TimeStamp contains date information. See [4 difference between Time and Timestamp in Java JDBC](http://javarevisited.blogspot.sg/2012/10/difference-between-javasqltime-date-timestamp-jdbc-interview-question.html) for more differences.

# [Why do we use a DataSource instead of a DriverManager?](http://stackoverflow.com/questions/15198319/why-do-we-use-a-datasource-instead-of-a-drivermanager)

<http://stackoverflow.com/questions/15198319/why-do-we-use-a-datasource-instead-of-a-drivermanager>

Better scalability and maintainence

For driver manager you need to know all the details (host, port, username, password, driver class) to connect to DB an to get connections. Externalizing those in a properties file doesn't change anything about the fact that you need to know them.

Using a Datasource you need to know only the JNDI name. The Appserver cares about the details and is not configured by the client application' vendor but by an admin where the application is hosted.

Scalability:

Suppose you need to create connections yourself, how would you deal with changing load, sometime you have 10 users sometime you have 1000, you can't just get a connection whenever you need one and later 'release' it so the Database server does not get out of connections, which leads you to connection pooling. DriverManager does not provide it, Datasource does.

If you are going to program a connection pool then you have to use DriverManager, otherwise go with Datasource.

# [Java JDBC DataSource Connection](http://stackoverflow.com/questions/17828158/java-jdbc-datasource-connection)

A DataSource allows getting a JDBC connection mostly from a pool of connections .A DataSource object represents a particular DBMS or some other data source, such as a file. If a company uses more than one data source, it will deploy a separate DataSource object for each of them. The DataSource interface is implemented by a driver vendor. You externalize DB connection properties file and fetch the object using JNDI . Using a Datasource you need to know only the JNDI name. The Application server cares about the details.

It can be implemented in three different ways:

1. A basic DataSource implementation produces standard Connection objects that are not pooled or used in a distributed transaction.
2. A DataSource implementation that supports connection pooling produces Connection objects that participate in connection pooling, that is, connections that can be recycled.
3. A DataSource implementation that supports distributed transactions produces Connection objects that can be used in a distributed transaction, that is, a transaction that accesses two or more DBMS servers.

Like, in Spring , you can configure the datasource in an xml file and then (1) either inject it into your bean , (2) get it from ApplicationContext

DataSource ds = (DataSource)ApplicationContextProvider.

getApplicationContext().getBean("myDataSource");

Connection c = ds.getConnection();

**Suggested Reading :**

1. [Connecting with DataSource Objects](http://docs.oracle.com/javase/tutorial/jdbc/basics/sqldatasources.html)
2. [Why do we use a DataSource instead of a DriverManager?](http://stackoverflow.com/questions/15198319/why-do-we-use-a-datasource-instead-of-a-drivermanager)
3. [Data access with JDBC](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/jdbc.html)
4. [How to retrieve DB connection using DataSource without JNDI?](http://stackoverflow.com/questions/15565689/how-to-retrieve-db-connection-using-datasource-without-jndi)
5. [Best way to manage DB connections without JNDI](http://stackoverflow.com/questions/3811385/best-way-to-manage-db-connections-without-jndi?rq=1)

# [How to retrieve DB connection using DataSource without JNDI?](http://stackoverflow.com/questions/15565689/how-to-retrieve-db-connection-using-datasource-without-jndi)

You use a connection pool library like [c3p0](http://www.mchange.com/projects/c3p0/) or [commons dbcp](http://sourceforge.net/projects/c3p0/).

C3P0

ComboPooledDataSource cpds = new ComboPooledDataSource();

cpds.setDriverClass( "org.postgresql.Driver" ); //loads the jdbc driver

cpds.setJdbcUrl( "jdbc:postgresql://localhost/testdb" );

cpds.setUser("dbuser");

cpds.setPassword("dbpassword");

Connection connection = cpds.getConnection();

DBCP

BasicDataSource ds= new BasicDataSource();

ds.setDriverClassName("org.postgresql.Driverr");

ds.setUrl("jdbc:postgresql://localhost/testdb");

ds.setUsername("dbuser");

ds.setPassword("dbpassword");

Connection connection = ds.getConnection();