Contents

[Implementations of the PlatformTransactionManager. What are the two interfaces used by PlatformTransactionManager. 2](#_Toc14321400)

[TransactionDefinition and TransactionStatus Inteface 2](#_Toc14321401)

[Transaction Isolation Levels 3](#_Toc14321402)

[To enable annotation support for transaction management in Spring using XMLconfiguration, we need to add the <tx:annotation-driven> tag in the XML configuration file. 4](#_Toc14321403)

[<tx:annotation-driven> tag 4](#_Toc14321404)

[Things To Remember 4](#_Toc14321405)

[Things To Remember 5](#_Toc14321406)

[Spring Manager Transactions for Different API 5](#_Toc14321407)

[How to call store procedure in Spring JDBC | Spring SimpleJdbcCall Example 5](#_Toc14321408)

[Usage of SqlParameterSource Spring Example 8](#_Toc14321409)

[MappingSqlQuery Spring Example 11](#_Toc14321410)

[SimpleJdbcTemplate Spring Example 11](#_Toc14321411)

[Example to Update Database in Spring JDBC 11](#_Toc14321412)

[Spring JDBC Integration Example| JdbcTemplate in Spring 11](#_Toc14321413)

[queryForInt and queryForObject in Spring JDBC 11](#_Toc14321414)

[SqlUpdate Spring JDBC Example 11](#_Toc14321415)

[NamedParameterJdbcTemplate Spring Example 11](#_Toc14321416)

[How to Get Auto Generated ID in Spring JDBC| Spring KeyHolder Example 11](#_Toc14321417)

[Spring JDBC 12](#_Toc14321418)

[Ho to implement save point. 13](#_Toc14321419)

[How does Spring @Transactional Really Work? 14](#_Toc14321420)

# Implementations of the PlatformTransactionManager. What are the two interfaces used by PlatformTransactionManager.

In Spring, the PlatformTransactionManager interface uses the TransactionDefinition and

TransactionStatus interfaces to create and manage transactions. The actual implementation of these

interfaces must have detailed knowledge of the transaction manager.

# TransactionDefinition and TransactionStatus Inteface

TransactionDefinition interface controls the properties of a transaction

The TransactionStatus interface is used to

control the transaction execution, more specifically to set the transaction result and to check whether the

transaction is completed or whether it is a new transaction.

Transactions have the four notoriously known ACID properties (atomicity, consistency, isolation, and

durability), and it is up to the transactional resources to maintain these aspects of a transaction. You cannot

control the atomicity, consistency, and durability of a transaction. However, you can control the transaction

propagation and timeout, as well as configure whether the transaction should be read-only and specify the

isolation level.

Spring encapsulates all these settings in a TransactionDefinition interface. This interface is used in

the core interface of the transaction support in Spring, which is the PlatformTransactionManager interface,

whose implementations perform transaction management on a specific platform, such as JDBC or JTA. The

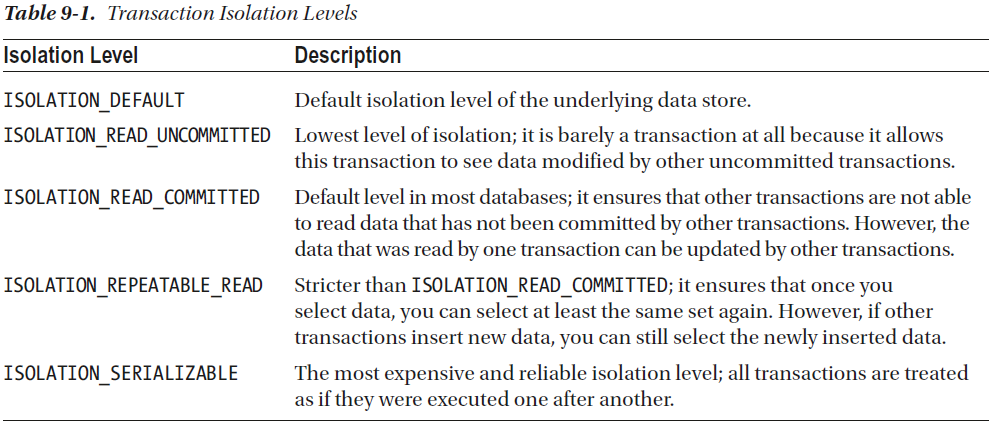
core method, PlatformTransactionManager.getTransaction(), takes a TransactionDefinition interface

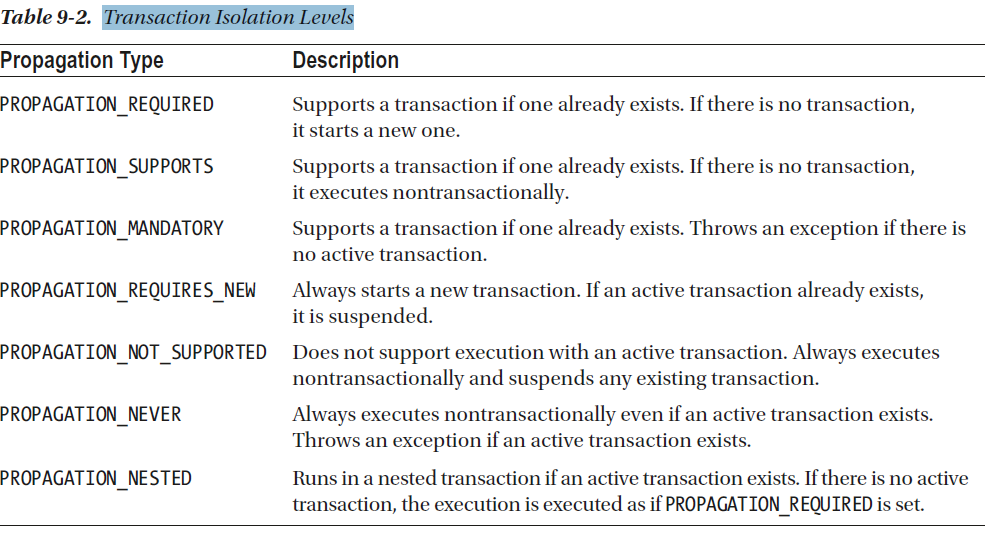
as an argument and returns a TransactionStatus interface. The TransactionStatus interface is used to

control the transaction execution, more specifically to set the transaction result and to check whether the

transaction is completed or whether it is a new transaction.

# Transaction Isolation Levels





# To enable annotation support for transaction management in Spring using XMLconfiguration, we need to add the <tx:annotation-driven> tag in the XML configuration file.

# <tx:annotation-driven> tag

The <tx:annotation-driven>

tag specifies that we are using annotations for transaction management.

This simple definition instructs Spring to look for a bean named transactionManager of type PlatformTransactionManager.

If the transaction bean is named differently, let’s say customTransactionManager, the element definition must be declared with the transaction-manager attribute that must receive as a value the name of the transaction management bean.

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

# Things To Remember

As discussed in Chapter 5, transaction management is a crosscutting concern and should not be coded

within the business logic. The most appropriate way to implement transaction management is to allow

developers to define transaction requirements in a declarative way and have frameworks such as Spring, JEE,

or AOP weave in the transaction processing logic on our behalf. In this chapter, we discuss how Spring helps

simplify the implementation of transaction-processing logic. Spring provides support for both declarative

and programmatic transaction management.

# Things To Remember

Programmatic transaction management: Even though programmatic transaction

management is not used very often, we explain how to use the Spring-provided

TransactionTemplate class, which gives you full control over the transaction

management code.

In professional environments, it is a common practice to separate the persistence configuration (DAO)

from the transactional configuration (service). That is why the contents of the XML introduced before were

split in the Java configuration into two configuration classes. DataJpaConfig, which was introduced before,

contains only data access beans, and ServicesConfig, which is depicted next, contains only transactional

management–related beans:

package com.apress.prospring5.ch9.config;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.persistence.EntityManagerFactory;

@Configuration

@EnableTransactionManagement

@ComponentScan(basePackages = "com.apress.prospring5.ch9")

public class ServicesConfig {

@Autowired EntityManagerFactory entityManagerFactory;

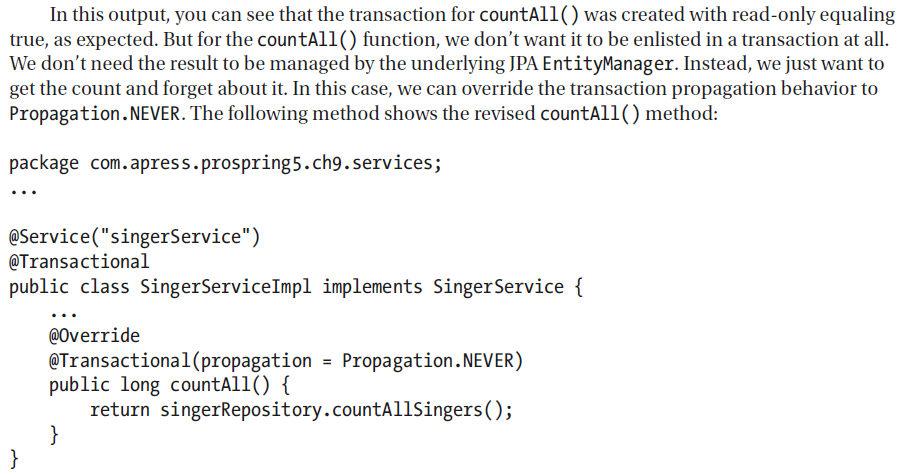
@Bean

public PlatformTransactionManager transactionManager() {

return new JpaTransactionManager(entityManagerFactory);

}

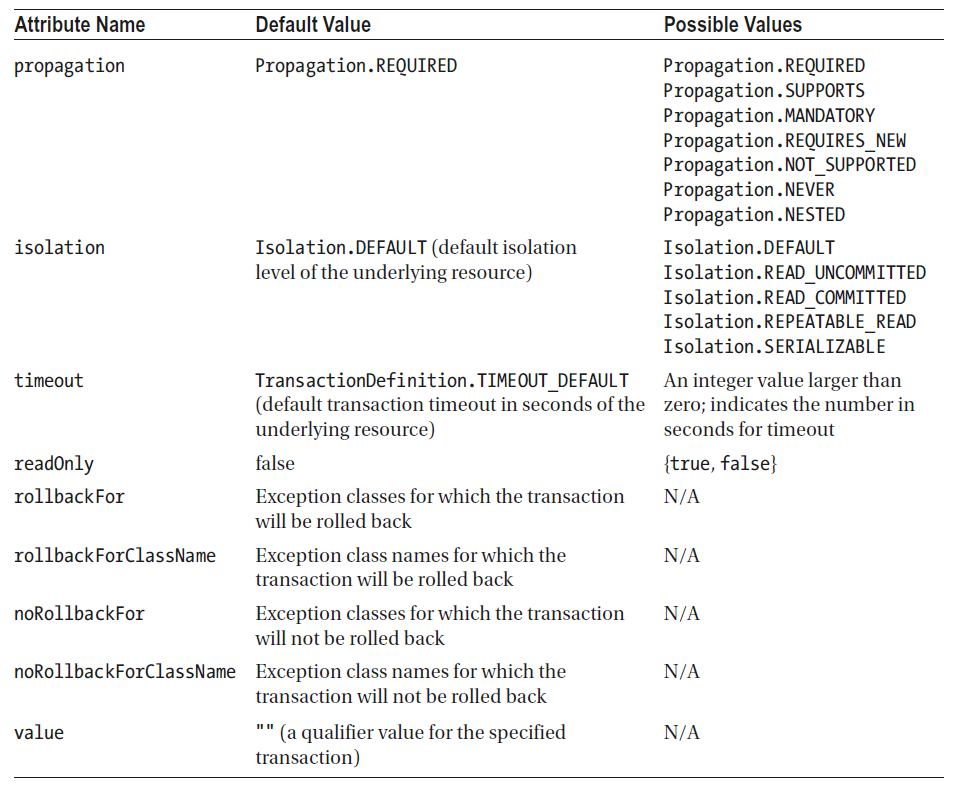
}



# Things To Remember

### Attributes for the @Transactional Annotation - Default Value Propagation.REQUIRED & Isolation.DEFAULT and

### TransactionDefinition.TIMEOUT\_DEFAULT(default transaction timeout in seconds of the underlying resource)



# Spring’s JpaTransactionManager doesn’t support a custom isolation level

Spring’s JpaTransactionManager doesn’t support a custom isolation level. Instead, it always uses the

default isolation level for the underlying data store. If you are using Hibernate as the JPA service provider, you

can use a workaround: extend the HibernateJpaDialect class to support a custom isolation level.

# **Spring Manager Transactions for Different API**

JTA, JDBC, HIBERNATE, JPA, JDO, JMS

# **How to call store procedure in Spring JDBC | Spring SimpleJdbcCall Example**

In this page, we will learn how to call a stored procedure in our spring application with JDBC. Spring JDBC provides SimpleJdbcCall that has execute() method. Execute method takes input IN parameter of stored procedure. SimpleJdbcCall.execute() returns a Map. And that map stores key as out parameter name and value is value returned by stored procedure. Find the example how to do it.

#### CompanyDao.java

package com.concretepage.dao;

import java.util.Map;

import javax.sql.DataSource;

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

import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;

import org.springframework.jdbc.core.namedparam.SqlParameterSource;

import org.springframework.jdbc.core.simple.SimpleJdbcCall;

import org.springframework.stereotype.Repository;

import com.concretepage.bean.Company;

@Repository

public class CompanyDao {

private SimpleJdbcCall simpleJdbcCall;

@Autowired

public void setDataSource(DataSource dataSource) {

**this.simpleJdbcCall = new SimpleJdbcCall(dataSource).withProcedureName("read\_company");**

}

public Company getCompnay(int id){

SqlParameterSource in = new MapSqlParameterSource().addValue("in\_id", id);

Map<String,Object> out = simpleJdbcCall.execute(in);

Company comp = new Company();

comp.setName((String)out.get("out\_name"));

comp.setLocation((String)out.get("out\_location"));

comp.setNoOfEmp(Integer.parseInt(String.valueOf(out.get("out\_no\_emp"))));

return comp;

}

}

#### Company.java

package com.concretepage.bean;

public class Company {

private long id;

private String name;

private String location;

private int noOfEmp;

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getLocation() {

return location;

}

public void setLocation(String location) {

this.location = location;

}

public int getNoOfEmp() {

return noOfEmp;

}

public void setNoOfEmp(int noOfEmp) {

this.noOfEmp = noOfEmp;

}

}

#### SpringTest.java

package com.concretepage;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.concretepage.bean.Company;

import com.concretepage.dao.CompanyDao;

public class SpringTest {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");

CompanyDao companyDao = (CompanyDao)context.getBean("companyDao");

Company comp = companyDao.getCompnay(1);

System.out.println(comp.getLocation()+" "+comp.getName()+" "+comp.getNoOfEmp());

}

}

#### Stored procedure: read\_company

CREATE PROCEDURE `read\_company` (

IN in\_id INTEGER,

OUT out\_name VARCHAR(100),

OUT out\_location VARCHAR(100),

OUT out\_no\_emp INTEGER)

BEGIN

SELECT name, location, no\_of\_emp

INTO out\_name, out\_location, out\_no\_emp

FROM company where id = in\_id;

END;

#### Table Schema: company

CREATE TABLE `company` (

`id` INT(11) NOT NULL AUTO\_INCREMENT,

`name` VARCHAR(50) NULL DEFAULT NULL,

`location` VARCHAR(50) NULL DEFAULT NULL,

`no\_of\_emp` INT(11) NULL DEFAULT NULL,

PRIMARY KEY (`id`)

)

#### Spring.xml

<?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: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/context

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

<bean id="companyDao" class="com.concretepage.dao.CompanyDao">

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

</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/test"/>

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

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

</bean>

</beans>

# **Usage of SqlParameterSource Spring Example**

SqlParameterSource in spring is used to insert rows in table. SqlParameterSource is more efficient than plane map approach. Java bean which is mapped to table, can directly be used to insert values. SqlParameterSource has two implementation BeanPropertySqlParameterSource and MapSqlParameterSource. We will show the uses of both classes approach.

#### Using BeanPropertySqlParameterSource

#### CompanyDao.java

package com.concretepage.dao;

import javax.sql.DataSource;

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

import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;

import org.springframework.jdbc.core.namedparam.SqlParameterSource;

import org.springframework.jdbc.core.simple.SimpleJdbcInsert;

import org.springframework.stereotype.Repository;

import com.concretepage.bean.Company;

@Repository

public class CompanyDao {

private SimpleJdbcInsert insertCompany;

@Autowired

public void setDataSource(DataSource dataSource) {

this.insertCompany =

**new SimpleJdbcInsert(dataSource).withTableName("company").usingGeneratedKeyColumns("id");**

}

public void addCompnay(Company comp){

**SqlParameterSource parameters = new BeanPropertySqlParameterSource(comp);**

**Number genId = insertCompany.executeAndReturnKey(parameters);**

**comp.setId(genId.longValue());**

}

}

#### Using MapSqlParameterSource

#### CompanyDao.java

package com.concretepage.dao;

import javax.sql.DataSource;

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

import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;

import org.springframework.jdbc.core.namedparam.SqlParameterSource;

import org.springframework.jdbc.core.simple.SimpleJdbcInsert;

import org.springframework.stereotype.Repository;

import com.concretepage.bean.Company;

@Repository

public class CompanyDao {

private SimpleJdbcInsert insertCompany;

@Autowired

public void setDataSource(DataSource dataSource) {

this.insertCompany = new SimpleJdbcInsert(dataSource).withTableName("company").usingGeneratedKeyColumns("id");

}

public void addCompnay(Company comp){

SqlParameterSource parameters =

**new MapSqlParameterSource().addValue("name", comp.getName())**

**.addValue("location",comp.getId()).addValue("no\_of\_emp",comp.getNoOfEmp());**

Number genId = insertCompany.executeAndReturnKey(parameters);

comp.setId(genId.longValue());

}

}

#### Company.java

package com.concretepage.bean;

public class Company {

private long id;

private String name;

private String location;

private int noOfEmp;

public Company(String name, String location, int noOfEmp){

this.name = name;

this.location = location;

this.noOfEmp = noOfEmp;

}

public String getName() {

return name;

}

public String getLocation() {

return location;

}

public int getNoOfEmp() {

return noOfEmp;

}

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

}

#### SpringTest.java

package com.concretepage;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.concretepage.bean.Company;

import com.concretepage.dao.CompanyDao;

public class SpringTest {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");

CompanyDao companyDao = (CompanyDao)context.getBean("companyDao");

Company comp = new Company("Techwox", "Varanasi", 100);

companyDao.addCompnay(comp);

System.out.println("Generated Id:"+comp.getId());

}

}

#### Spring.xml

<?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: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/context

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

<bean id="companyDao" class="com.concretepage.dao.CompanyDao">

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

</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/test"/>

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

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

</bean>

</beans>

#### Table Schema: company

CREATE TABLE `company` (

`id` INT(11) NOT NULL AUTO\_INCREMENT,

`name` VARCHAR(50) NULL DEFAULT NULL,

`location` VARCHAR(50) NULL DEFAULT NULL,

`no\_of\_emp` INT(11) NULL DEFAULT NULL,

PRIMARY KEY (`id`)

)

### MappingSqlQuery Spring Example

<https://www.concretepage.com/spring/mappingsqlquery-spring-example>

### SimpleJdbcTemplate Spring Example

<https://www.concretepage.com/spring/simplejdbctemplate-spring-example>

### Example to Update Database in Spring JDBC

<https://www.concretepage.com/spring/example-to-update-database-in-spring-jdbc>

### Spring JDBC Integration Example| JdbcTemplate in Spring

<https://www.concretepage.com/spring/jdbc-template-spring-jdbc-integration-example>

### queryForInt and queryForObject in Spring JDBC

<https://www.concretepage.com/spring/queryforint-and-queryforobject-in-spring-jdbc>

### SqlUpdate Spring JDBC Example

<https://www.concretepage.com/spring/sqlupdate-spring-jdbc-example>

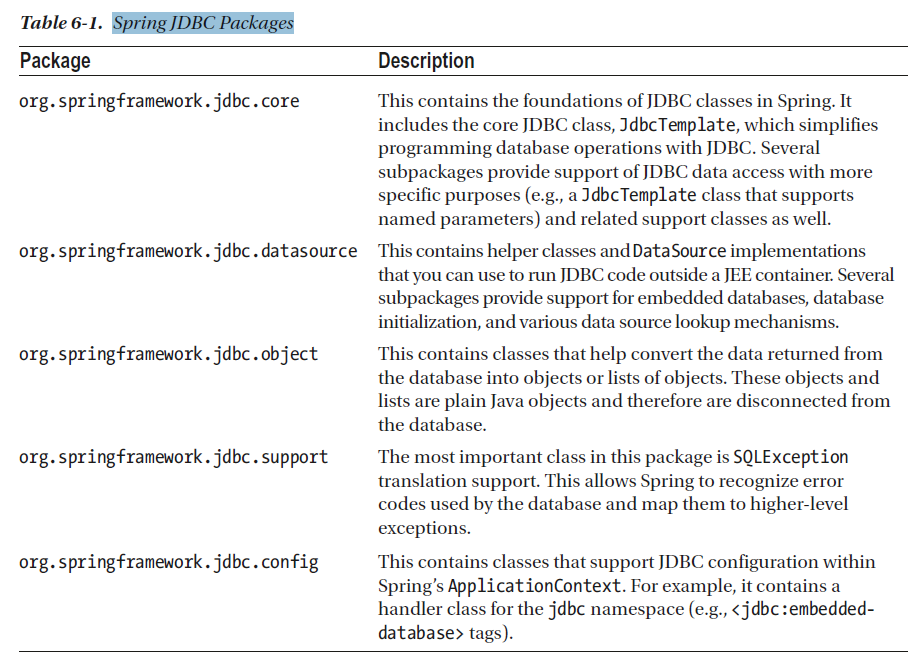
### NamedParameterJdbcTemplate Spring Example

# How to Get Auto Generated ID in Spring JDBC| Spring KeyHolder Example

<https://www.concretepage.com/spring/how-to-get-auto-generated-id-in-spring-jdbc>

# **Spring JDBC**

#### Spring JDBC Packages



#### DriverManagerDataSource Database Connections and DataSources

You can use Spring to manage the database connection for you by providing a bean that implements javax.

sql.DataSource. The difference between a DataSource and a Connection is that a DataSource provides and

manages Connections.

DriverManagerDataSource (under the package org.springframework.jdbc.datasource) is the

simplest implementation of a DataSource. By looking at the class name, you can guess that it simply calls

DriverManager to obtain a connection. The fact that DriverManagerDataSource doesn’t support database

connection pooling makes this class unsuitable for anything other than testing. The configuration of

DriverManagerDataSource is quite simple, as you can see in the following code snippet; you just need to

supply the driver class name, a connection URL, a username, and a password

# Ho to implement save point.

The Connection.setSavepoint() method is used to set a savepoint object within the current transaction.

The following example demonstrates how the Savepoint object is used in JDBC application.

import java.sql.Connection;

import java.sql.Date;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import java.sql.Savepoint;

import java.text.DateFormat;

import java.text.ParseException;

import java.text.SimpleDateFormat;

/\*\*

\* @author imssbora

\*/

public class JDBCSavepointExample {

private static final String INSERT\_SQL = "INSERT INTO employee "

+ "(EMP\_ID, NAME, DOB) VALUES (?,?,?)";

public static void main(String[] args) {

String jdbcUrl = "jdbc:mysql://localhost:3306/BORAJI";

String username = "root";

String password = "admin";

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

try (Connection conn = DriverManager.getConnection(jdbcUrl, username, password);) {

// Disable auto commit mode

conn.setAutoCommit(false);

try (PreparedStatement insertStmt = conn.prepareStatement(INSERT\_SQL);) {

// Insert 1st record

insertStmt.setInt(1, 1);

insertStmt.setString(2, "Michael");

insertStmt.setDate(3, new Date(dateFormat.parse("1995-07-01").getTime()));

insertStmt.executeUpdate();

// Insert 2st record

insertStmt.setInt(1, 2);

insertStmt.setString(2, "Sunil");

insertStmt.setDate(3, new Date(dateFormat.parse("1988-03-22").getTime()));

insertStmt.executeUpdate();

// Insert 3st record

insertStmt.setInt(1, 3);

insertStmt.setString(2, "Mike");

insertStmt.setDate(3, new Date(dateFormat.parse("1980-05-12").getTime()));

insertStmt.executeUpdate();

// Create Savepoint

Savepoint savepoint = conn.setSavepoint();

// Insert 4st record

insertStmt.setInt(1, 4);

insertStmt.setString(2, "Manish");

insertStmt.setDate(3, new Date(dateFormat.parse("1992-01-21").getTime()));

insertStmt.executeUpdate();

// Insert 5st record

insertStmt.setInt(1, 5);

insertStmt.setString(2, "Albert");

insertStmt.setDate(3, new Date(dateFormat.parse("1972-07-05").getTime()));

insertStmt.executeUpdate();

// Rollback to savepoint

conn.rollback(savepoint);

// Commit statement

conn.commit();

System.out.println("Transaction is commited successfully.");

} catch (SQLException | ParseException e) {

e.printStackTrace();

if (conn != null) {

try {

// Roll back transaction

System.out.println("Transaction is being rolled back.");

conn.rollback();

} catch (Exception ex) {

ex.printStackTrace();

}

}

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

# How does Spring @Transactional Really Work?

<https://www.javacodegeeks.com/2014/06/how-does-spring-transactional-really-work.html>

|  |
| --- |
| What is the role of TransactionDefinition interface? |
| Ans: a. Isolation b. Propagation  c. Timeout  d. Read-only status |
|  |
| How can we roll back a declarative transaction? |
| Ans: We can use rollback-for and no-rollback-for attributes with transactional definition. |
|  |
| How many types of isolation are there? |
| Ans: a. ISOLATION\_DEFAULT: default isolation.  b. ISOLATION\_READ\_COMMITTED: dirty reads are prevented, non-repeatable and phantom reads are allowed. c. ISOLATION\_READ\_UNCOMMITTED : dirty reads are allowed, no-repeatable and phantom reads are allowed.  d. ISOLATION\_REPEATABLE\_READ: dirty reads and non-repeatable reads are prevented but phantom reads are allowed.  e. ISOLATION\_SERIALIZABLE : dirty , non- repeatable reads and phantom reads are prevented. |
|  |
| Qns-5: How many types of Propagation are there? |
| Ans: Find the Propagation type. a. PROPAGATION\_MANDATORY : supports current transaction and throws exception if no transaction available.  b. PROPAGATION\_NESTED : runs with nested transaction  c. PROPAGATION\_NEVER : does not run with current transaction and throws exception if current transaction exits.  d. PROPAGATION\_NOT\_SUPPORTED : runs non -transactionaly and does not support current transaction.  e. PROPAGATION\_REQUIRED : runs with current transaction and create one if does not exist.  f. PROPAGATION\_REQUIRES\_NEW : creates new transaction and suspends if exits any.  g. PROPAGATION\_SUPPORTS: runs current transaction and runs non -transactionaly |