[General Questions 1](#_Toc68179397)

[1) What spring jdbc does? 2](#_Toc68179398)

[2) Different approaches to access database? 2](#_Toc68179399)

[3) Spring jdbc package hierarchy 2](#_Toc68179400)

[4) Explain configuration for Datasource and jdbcTemplate? 3](#_Toc68179401)

[5) Explain jdbc Template configuration? 5](#_Toc68179402)

[6) Explain jdbcTemplate? 8](#_Toc68179403)

[7) Explain NamedParameterJdbcTemplate 11](#_Toc68179404)

[8) SimpleJdbcTemplate? 13](#_Toc68179405)

[9) Explain simpleJdbcInsert? 13](#_Toc68179406)

[10) How to retrive autogenerated key using simpleJdbcInsert? 14](#_Toc68179407)

[11) SimpleJdbcInsert for limited columns? 15](#_Toc68179408)

[12) SQLParameterSource 16](#_Toc68179409)

[13) Explain SimpleJdbcCall? 16](#_Toc68179410)

[14) Calling function withsimpleJdbcCall? 18](#_Toc68179411)

[15) How to return cursor from procedure? 18](#_Toc68179412)

[16) Different ways to configure datasource using java configuration? 19](#_Toc68179413)

[17) Datasource config using bean.xml 20](#_Toc68179414)

[18) Datasource using spring boot- 20](#_Toc68179415)

[19) Datasource and jdbcTemplate both in xml, old way? 20](#_Toc68179416)

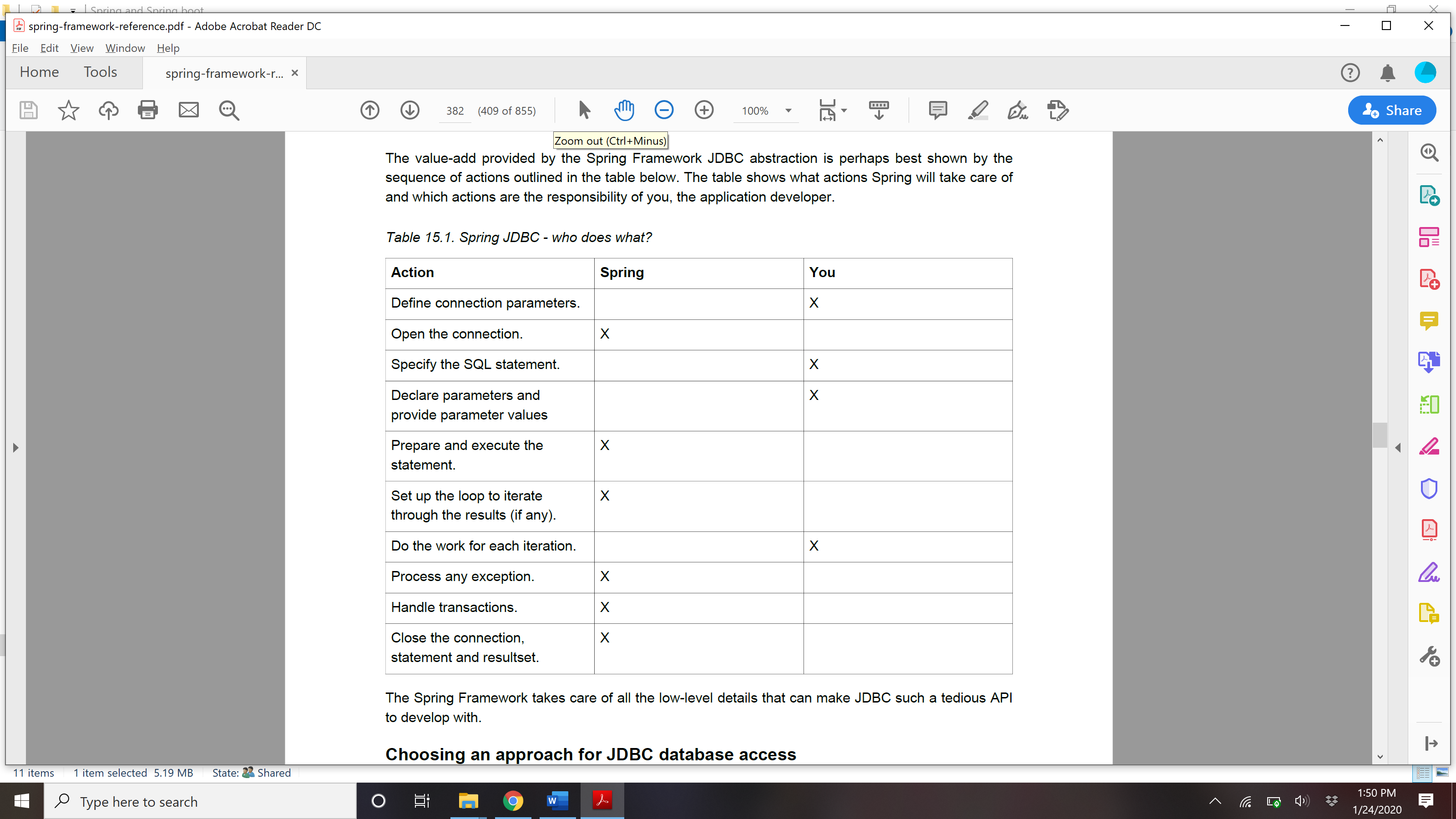
[20) Spring boot and jdbc? 21](#_Toc68179417)

[21) Spring jdbc vs Spring boot jdbc 21](#_Toc68179418)

[REFERENCES: 22](#_Toc68179419)

# General Questions

## What spring jdbc does?



## Different approaches to access database?

### ***JdbcTemplate***is the classic Spring JDBC approach and the most popular. This "lowest level"approach and all others use a JdbcTemplate under the covers.

### • ***NamedParameterJdbcTemplate***wraps a JdbcTemplate to provide named parameters instead of the traditional JDBC "?" placeholders. This approach provides better documentation and ease of use

### when you have multiple parameters for an SQL statement.

### ***SimpleJdbcInsert and SimpleJdbcCall* optimize** database metadata to limit the amount of necessary configuration. This approach simplifies coding so that you only need to provide the name of the table or procedure and provide a map of parameters matching the column names. This only works if the database provides adequate metadata. If the database doesn’t provide this metadata, you will have to provide explicit configuration of the parameters.

## Spring jdbc package hierarchy

### Org.springframework.jdbc.core -> JdbcTemplate, NamedJdbcTemplate, SimpleJdbcInsert, simpleJdbcCall

1. Org.springframework.jdbc.support-> All exceptions
2. Org.springframework.jdbc.datasources ->implementation of datasources
3. Org.springframework.jdbc.object-> contains classes that represent RDBMS

queries, updates, and stored procedures as thread-safe, reusable objects.

## Explain configuration for Datasource and jdbcTemplate?

### Spring obtains a connection to the database through a DataSource. A DataSource is part of the JDBC specification and is a generalized connection factory. It allows a container or a framework to hide connection pooling and transaction management issues from the application code. As a developer, you need not know details about how to connect to the database; that is the responsibility of the administrator that sets up the datasource. You most likely fill both roles as you develop and test code, but you do not

### necessarily have to know how the production data source is configured.

### When using Spring’s JDBC layer, you obtain a data source from JNDI or you configure your own with a connection pool implementation provided by a third party. Popular implementations are Apache Jakarta Commons DBCP and C3P0. Implementations in the Spring distribution are meant only for testing

### purposes and do not provide pooling. This section uses Spring’s DriverManagerDataSource implementation, and several additional

### implementations are covered later.

### **Note**

### Only use the DriverManagerDataSource class should only be used for testing purposes since it does not provide pooling and will perform poorly when multiple requests for a connection are made.

### You obtain a connection with DriverManagerDataSource as you typically obtain a JDBC connection. Specify the fully qualified classname of the JDBC driver so that the DriverManager can load the driver

### class. Next, provide a URL that varies between JDBC drivers. (Consult the documentation for your driverfor the correct value.) Then provide a username and a password to connect to the database. Here is an

### example of how to configure a DriverManagerDataSource in Java code:

DriverManagerDataSource dataSource = **new** DriverManagerDataSource();

dataSource.setDriverClassName(***"org.hsqldb.jdbcDriver"***);

dataSource.setUrl(***"jdbc:hsqldb:hsql://localhost:"***);

dataSource.setUsername(***"sa"***);

dataSource.setPassword(***""***);

OR XML-----------

<bean id=***"dataSource"* class**=***"org.springframework.jdbc.datasource.DriverManagerDataSource"***>

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

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

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

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

</bean>

<context:property-placeholder location=***"jdbc.properties"***/>

**DBCP CONFIG:**

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

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

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

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

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

</bean>

<context:property-placeholder location=***"jdbc.properties"***/>

**C3P0 configuration:**

<bean id=***"dataSource"* class**=***"com.mchange.v2.c3p0.ComboPooledDataSource"*** destroy-method=***"close"***>

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

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

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

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

</bean>

<context:property-placeholder location=***"jdbc.properties"***/>

## Explain jdbc Template configuration?

### A common practice when using the JdbcTemplate class (and the associated

### NamedParameterJdbcTemplate classes) is to configure a DataSource in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes; the JdbcTemplate is created in the setter for the DataSource. This leads to DAOs that look in part like the following:

JAVA:

**public class** JdbcCorporateEventDao **implements** CorporateEventDao {

**private** JdbcTemplate jdbcTemplate;

**public void** setDataSource(DataSource dataSource) {

**this.jdbcTemplate = new JdbcTemplate(dataSource);**

}

*// JDBC-backed implementations of the methods on the CorporateEventDao follow...*

}

-------------------------------- corresponding 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.xsd**

**http://www.springframework.org/schema/context**

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

**<bean id**=**"corporateEventDao" class**=**"com.example.JdbcCorporateEventDao">**

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

**</bean>**

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

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

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

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

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

**</bean>**

**<context:property-placeholder location**=**"jdbc.properties"/>**

**</beans>**

**2nd way : WITH ANNOTATION**

**@Repository**

**public class** JdbcCorporateEventDao **implements** CorporateEventDao {

**private** JdbcTemplate jdbcTemplate;

**@Autowired**

**public void** setDataSource(DataSource dataSource) {

**this.jdbcTemplate = new JdbcTemplate(dataSource);**

}

*// JDBC-backed implementations of the methods on the CorporateEventDao follow...*

}

Corresponding 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.xsd**

**http://www.springframework.org/schema/context**

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

*<!-- Scans within the base package of the application for @Component classes to configure as beans*

*-->*

**<context:component-scan base-package**=**"org.springframework.docs.test" />**

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

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

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

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

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

**</bean>**

## Explain jdbcTemplate?

### The JdbcTemplate class is the central class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors such as forgetting to close the connection.

### It performs the basic tasks of the core JDBC workflow such as statement creation and execution, leaving application code to provide SQL and extract results.

### **The JdbcTemplate class executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values.**

### It also catches JDBC exceptions and translates them to the generic, more

### informative, exception hierarchy defined in the org.springframework.dao package.

\*\* All SQL issued by this class is logged at the DEBUG level

**Querying (SELECT) - this.jdbcTemplate.queryForObject**

Here is a simple query for getting the number of rows in a relation:

**int** rowCount = **this.jdbcTemplate.queryForObject**(***"select count(\*) from t\_actor"***, Integer.**class**);

A simple query using a bind variable:

**int** countOfActorsNamedJoe = **this**.jdbcTemplate.queryForObject(

***"select count(\*) from t\_actor where first\_name = ?"***, Integer.**class**, ***"Joe"***);

Querying for a String:

String lastName = **this**.jdbcTemplate.queryForObject(

***"select last\_name from t\_actor where id = ?"***,

**new** Object[]{1212L}, String.**class**);

Querying and populating a *single* domain object:

Actor actor = **this**.jdbcTemplate.queryForObject(

***"select first\_name, last\_name from t\_actor where id = ?"***,

**new** Object[]{1212L},

**new** RowMapper<Actor>() {

**public** Actor mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

Actor actor = **new** Actor();

actor.setFirstName(rs.getString(***"first\_name"***));

actor.setLastName(rs.getString(***"last\_name"***));

**return** actor;

}

});

QUERY multiple rows – **this**.jdbcTemplate.query

List<Actor> actors = **this**.jdbcTemplate.query(

***"select first\_name, last\_name from t\_actor"***,

**new** RowMapper<Actor>() {

**public** Actor mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

Actor actor = **new** Actor();

actor.setFirstName(rs.getString(***"first\_name"***));

actor.setLastName(rs.getString(***"last\_name"***));

**return** actor;

}

});

We can create separate class of RowMapper:

**public** List<Actor> findAllActors() {

**return this**.jdbcTemplate.query( ***"select first\_name, last\_name from t\_actor"***, **new** ActorMapper());

}

**private static final class** ActorMapper **implements** RowMapper<Actor> {

**public** Actor mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

Actor actor = **new** Actor();

actor.setFirstName(rs.getString(***"first\_name"***));

actor.setLastName(rs.getString(***"last\_name"***));

**return** actor;

}

}

**Updating (INSERT/UPDATE/DELETE) with jdbcTemplate**

You use the update(..) method to perform insert, update and delete operations. Parameter values

are usually provided as var args or alternatively as an object array.

**this**.jdbcTemplate.update(

***"insert into t\_actor (first\_name, last\_name) values (?, ?)"***,

***"Leonor"***, ***"Watling"***);

**this**.jdbcTemplate.update(

***"update t\_actor set last\_name = ? where id = ?"***,

***"Banjo"***, 5276L);

**this**.jdbcTemplate.update(

***"delete from actor where id = ?"***,

Long.valueOf(actorId));

**Other jdbcTemplate operations**

You can use the execute(..) method to execute any arbitrary SQL, and as such the method is often used for DDL statements. It is heavily overloaded with variants taking callback interfaces, binding variable arrays, and so on.

**this**.jdbcTemplate.execute(***"create table mytable (id integer, name varchar(100))"***);

The following example invokes a simple stored procedure. More sophisticated stored procedure support is covered later.

**this**.jdbcTemplate.update(

***"call SUPPORT.REFRESH\_ACTORS\_SUMMARY(?)"***,

Long.valueOf(unionId));

\*\*\*\*\*\* Instances of the JdbcTemplate class are *threadsafe once configured*. This is important because it means that you can configure a single instance of a JdbcTemplate and then safely inject this *shared* reference into multiple DAOs (or repositories). The JdbcTemplate is stateful, in that it maintains are rference to a DataSource, but this state is *not* conversational state.

## Explain NamedParameterJdbcTemplate

### The NamedParameterJdbcTemplate class adds support for programming JDBC statements using named parameters, as opposed to programming JDBC statements using only classic placeholder (

### '?') arguments.

### The NamedParameterJdbcTemplate class wraps a JdbcTemplate, and delegates to the wrapped JdbcTemplate to do much of its work.

### *// some JDBC-backed DAO class...*

### **private** NamedParameterJdbcTemplate namedParameterJdbcTemplate;

### **public void** setDataSource(DataSource dataSource) {

### **this**.namedParameterJdbcTemplate = **new** NamedParameterJdbcTemplate(dataSource);

### 3e`}

### **public int** countOfActorsByFirstName(String firstName) {

### String sql = ***"select count(\*) from T\_ACTOR where first\_name = :first\_name"***;

### SqlParameterSource namedParameters = **new** MapSqlParameterSource(***"first\_name"***, firstName);

### **return this**.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.**class**);

### }

### Notice the use of the named parameter notation in the value assigned to the sql variable,and the corresponding value that is plugged into the namedParameters variable (of type MapSqlParameterSource).

### Alternatively, you can pass along named parameters and their corresponding values toa NamedParameterJdbcTemplate instance by using the Map-based style. The remainingmethods exposed by the NamedParameterJdbcOperations and implemented by the

### NamedParameterJdbcTemplate class follow a similar pattern and are not covered here.

### The following example shows the use of the Map-based style.

### Spring Framework Reference Documentation

### 5.0.0.M1 Spring Framework 389

### *// some JDBC-backed DAO class...*

### **private** NamedParameterJdbcTemplate namedParameterJdbcTemplate;

### **public void** setDataSource(DataSource dataSource) {

### **this**.namedParameterJdbcTemplate = **new** NamedParameterJdbcTemplate(dataSource);

### }

### **public int** countOfActorsByFirstName(String firstName) {

### String sql = ***"select count(\*) from T\_ACTOR where first\_name = :first\_name"***;

### Map<String, String> namedParameters = Collections.singletonMap(***"first\_name"***, firstName);

### **return this**.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.**class**);

### }

### One nice feature related to the NamedParameterJdbcTemplate (and existing in the same Java package) is the SqlParameterSource interface. You have already seen an example of an implementation of this interface in one of the previous code snippet (the MapSqlParameterSource class).

### An SqlParameterSource is a source of named parameter

### values to a NamedParameterJdbcTemplate. The MapSqlParameterSource class is a very simple implementation that is simply an adapter around a java.util.Map, where the keys are the parameter names and the values are the parameter values.

### Another SqlParameterSource implementation is the BeanPropertySqlParameterSource class.

### This class wraps an arbitrary JavaBean (that is, an instance of a class that adheres to the JavaBeanconventions), and uses the properties of the wrapped JavaBean as the source of named parameter

### values.

### **public class** Actor {

### **private** Long id;

### **private** String firstName;

### **private** String lastName;

### **public** String getFirstName() {

### **return this**.firstName;

### }

### **public** String getLastName() {

### **return this**.lastName;

### }

### **public** Long getId() {

### **return this**.id;

### }

### *// setters omitted...*

### }

### Spring Framework Reference Documentation

### 5.0.0.M1 Spring Framework 390

### *// some JDBC-backed DAO class...*

### **private** NamedParameterJdbcTemplate namedParameterJdbcTemplate;

### **public void** setDataSource(DataSource dataSource) {

### **this**.namedParameterJdbcTemplate = **new** NamedParameterJdbcTemplate(dataSource);

### }

### **public int** countOfActors(Actor exampleActor) {

### *// notice how the named parameters match the properties of the above 'Actor' class*

### String sql = ***"select count(\*) from T\_ACTOR where first\_name = :firstName and last\_name = :lastName"***;

### SqlParameterSource namedParameters = **new** BeanPropertySqlParameterSource(exampleActor);

### **return this**.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.**class**);

### }

### Remember that the NamedParameterJdbcTemplate class *wraps* a classic JdbcTemplate template; if you need access to the wrapped JdbcTemplate instance to access functionality only present in the JdbcTemplate class, you can use the getJdbcOperations() method to access the wrapped

### JdbcTemplate through the JdbcOperations interface.

## SimpleJdbcTemplate?

### This was introduced in spring 3 to support varagrgs of java 5, but has ben deprecated in spring 5 because Jdbctemplate abd NamedJdbcTEmeplate provides all those functionalities.

## Explain simpleJdbcInsert?

### The SimpleJdbcInsert and SimpleJdbcCall classes provide a simplified configuration by taking advantage of database metadata that can be retrieved through the JDBC driver. This means there is less to configure up front, although you can override or turn off the metadata processing if you prefer

### to provide all the details in your code.

### \*\*You should know table metadata for this

### .

**public class** JdbcActorDao **implements** ActorDao {

**private** JdbcTemplate jdbcTemplate;

**private** SimpleJdbcInsert insertActor;

**public void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

**this**.insertActor = **new** SimpleJdbcInsert(dataSource).withTableName(***"t\_actor"***);

}

**public void** add(Actor actor) {

Map<String, Object> parameters = **new** HashMap<String, Object>(3);

parameters.put(***"id"***, actor.getId());

parameters.put(***"first\_name"***, actor.getFirstName());

parameters.put(***"last\_name"***, actor.getLastName());

insertActor.execute(parameters);

}

*// ... additional methods*

}

The execute method used here takes a plain java.utils.Map as its only parameter. The important thing to note here is that the keys used for the Map must match the column names of the table as defined in the database. This is because we read the metadata in order to construct the actual insert statement.

## How to retrive autogenerated key using simpleJdbcInsert?

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcInsert insertActor;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

this.insertActor = new SimpleJdbcInsert(dataSource)

.withTableName("t\_actor")

.usingGeneratedKeyColumns("id");

}

public void add(Actor actor) {

Map<String, Object> parameters = new HashMap<String, Object>(2);

parameters.put("first\_name", actor.getFirstName());

parameters.put("last\_name", actor.getLastName());

Number newId = insertActor.executeAndReturnKey(parameters);

actor.setId(newId.longValue());

}

// ... additional methods

}

## SimpleJdbcInsert for limited columns?

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcInsert insertActor;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

this.insertActor = new SimpleJdbcInsert(dataSource)

.withTableName("t\_actor")

.usingColumns("first\_name", "last\_name")

.usingGeneratedKeyColumns("id");

}

public void add(Actor actor) {

Map<String, Object> parameters = new HashMap<String, Object>(2);

parameters.put("first\_name", actor.getFirstName());

parameters.put("last\_name", actor.getLastName());

Number newId = insertActor.executeAndReturnKey(parameters);

actor.setId(newId.longValue());

}

//

1. We can also provice SQLParameterREsouce instead of map.

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcInsert insertActor;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

this.insertActor = new SimpleJdbcInsert(dataSource)

.withTableName("t\_actor")

.usingGeneratedKeyColumns("id");

}

public void add(Actor actor) {

SqlParameterSource parameters = new BeanPropertySqlParameterSource(actor);

Number newId = insertActor.executeAndReturnKey(parameters);

actor.setId(newId.longValue());

}

// ... additional methods

}

## SQLParameterSource

SQLParameterSpurce

Map

BeanNamedSQLParameterSource

## Explain SimpleJdbcCall?

### The SimpleJdbcCall class leverages metadata in the database to look up names of in and outparameters, so that you do not have to declare them explicitly. You can declare parameters if you prefer to do that, or if you have parameters such as ARRAY or STRUCT that do not have an automatic mapping

### to a Java class. The first example shows a simple procedure that returns only scalar values in VARCHAR and DATE format from a MySQL database. The example procedure reads a specified actor entry and returns first\_name, last\_name, and birth\_date columns in the form of out parameters

**CREATE PROCEDURE** read\_actor (

**IN** in\_id **INTEGER**,

**OUT** out\_first\_name **VARCHAR**(100),

**OUT** out\_last\_name **VARCHAR**(100),

**OUT** out\_birth\_date **DATE**)

**BEGIN**

**SELECT** first\_name, last\_name, birth\_date

**INTO** out\_first\_name, out\_last\_name, out\_birth\_date

**FROM** t\_actor **where** id = in\_id;

**END**;

The SimpleJdbcCall is declared in a similar manner to the SimpleJdbcInsert. You should instantiate and configure the class in the initialization method of your data access layer.

**public class** JdbcActorDao **implements** ActorDao {

**private** JdbcTemplate jdbcTemplate;

**private** SimpleJdbcCall procReadActor;

**public void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

**this**.procReadActor = **new** SimpleJdbcCall(dataSource)

.withProcedureName(***"read\_actor"***);

}

**public** Actor readActor(Long id) {

SqlParameterSource in = **new** MapSqlParameterSource()

.addValue(***"in\_id"***, id);

Map out = procReadActor.execute(in);

Actor actor = **new** Actor();

actor.setId(id);

actor.setFirstName((String) out.get(***"out\_first\_name"***));

actor.setLastName((String) out.get(***"out\_last\_name"***));

actor.setBirthDate((Date) out.get(***"out\_birth\_date"***));

**return** actor;

}

*// ... additional methods*

}

The code you write for the execution of the call involves creating an SqlParameterSource containing the IN parameter. It’s important to match the name provided for the input value with that of the parameter name declared in the stored procedure. This is case insenstitive.

\*\*\* The execute method takes the IN parameters and returns a Map containing any out parameters

keyed by the name as specified in the stored procedure

## Calling function withsimpleJdbcCall?

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcCall funcGetActorName;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

jdbcTemplate.setResultsMapCaseInsensitive(true);

this.funcGetActorName = new SimpleJdbcCall(jdbcTemplate)

.withFunctionName("get\_actor\_name");

}

public String getActorName(Long id) {

SqlParameterSource in = new MapSqlParameterSource()

.addValue("in\_id", id);

String name = funcGetActorName.executeFunction(String.class, in);

return name;

}

## How to return cursor from procedure?

public class JdbcActorDao implements ActorDao {

private SimpleJdbcCall procReadAllActors;

public void setDataSource(DataSource dataSource) {

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

jdbcTemplate.setResultsMapCaseInsensitive(true);

this.procReadAllActors = new SimpleJdbcCall(jdbcTemplate)

.withProcedureName("read\_all\_actors")

.returningResultSet("actors",

BeanPropertyRowMapper.newInstance(Actor.class));

}

public List getActorsList() {

Map m = procReadAllActors.execute(new HashMap<String, Object>(0));

return (List) m.get("actors");

}

// ... additional methods

}

## Different ways to configure datasource using java configuration?

@Configuration

@ComponentScan("com.baeldung.jdbc")

public class SpringJdbcConfig {

    @Bean

    public DataSource mysqlDataSource() {

        DriverManagerDataSource dataSource = new DriverManagerDataSource();

        dataSource.setDriverClassName("com.mysql.jdbc.Driver");

        dataSource.setUrl("jdbc:<mysql://localhost:3306/springjdbc>");

        dataSource.setUsername("guest\_user");

        dataSource.setPassword("guest\_password");

        return dataSource;

    }

}

## Datasource config using bean.xml

<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/springjdbc>"/>

    <property name="username" value="guest\_user"/>

    <property name="password" value="guest\_password"/>

</bean>

## Datasource using spring boot-

spring.datasource.url=jdbc:mysql://localhost:3306/springjdbc

spring.datasource.username=guest\_user

spring.datasource.password=guest\_password

in application.properties

## Datasource and jdbcTemplate both in xml, old way?

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jtemplate" **class**="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
17. <constructor-arg ref="ds"></constructor-arg>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmpDao">
21. <constructor-arg>
22. <ref bean="jtemplate"/>
23. </constructor-arg>
24. </bean>
26. </beans>

## Spring boot and jdbc?

<https://mkyong.com/spring-boot/spring-boot-jdbc-examples/>

## Spring jdbc vs Spring boot jdbc

The functionality of Spring JDBC and Spring Boot JDBC are the same. Only the implementation is made simple. The following are the advantages of Spring Boot JDBC over Spring JDBC.

|  |  |
| --- | --- |
| **JDBC using Spring** | **JDBC using Spring Boot** |
| Multiple dependencies like spring-context, spring-jdbc need to be specified. | Only a single spring-boot starter dependency is required. |
| Necessary to create a database bean either using xml or javaconfig.  <bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource"> <property name="driverClass" value="org.hsqldb.jdbcDriver" /> <property name="url" value="jdbc:hsqldb:file:database.dat;shutdown=true" /> <property name="username" value="sa" /> <property name="password" value="" /> </bean> | Datasource bean gets initialized automatically if not mentioned explicitly. If user does not want this then it can be done by setting the property **spring.datasource.initialize** to false. |
| The Template beans PlatformTransactionManager, JdbcTemplate, NamedParameterJdbcTemplate must be registered | If the Template beans PlatformTransactionManager, JdbcTemplate, NamedParameterJdbcTemplate not registered, then Spring Boot will register them automatically. |
| If any db initialization scripts like dropping or creation of tables are created in sql file. This info needs to be given explicitly in the configuration. | Any db initialization scripts stored in schema-.sql gets executed automatically. |

# REFERENCES:

<https://www.javatpoint.com/spring-SimpleJdbcTemplate-example>

<https://www.baeldung.com/spring-jdbc-jdbctemplate>

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

<https://www.javainuse.com/spring/bootjdbc>