Spring JDBC

Spring JDBC is approach to access the DB applications in more effective way.

Classic Approach:

Classic approach resembles closely with java JDBC API approach only. There are 2 main aspects of programming are there

1. Classic approach works based on Call-back mechanism

2. Always in classic approach every operation will be performed in 2 phases.

2.1 prepare phase = prepare the statement and substitute the parameters

2.2 call-back back = execute the statement and collect the data from ResultSet

To use PreparedStatement in executing the above SQL query we need to write our code implementing 2 interfaces

interface PreparedStatementCreator {

PreparedStatement createPreparedStatement(Connection con);

}

interface PreparedStatementCallback {

Object doInPreparedStatement(PreparedStatement pstmt);

}

class ProductBo {

private int productNo;

private String productName;

private String description;

private String manufacturer;

private float price;

// accessors

}

class ProductDao {

private final String SQL\_GET\_PRODUCTS\_BY\_NM = "select \* from product where product\_nm like ?";

private JdbcTemplate jdbcTemplate;

public ProductDao(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public List<ProductBo> getProductsByProductName(String productName) {

List<ProductBo> productBos = null;

GetProductsByNamePreparedStatementCreator getProductsByNamePreparedStatementCreator = null;

GetProductsByProductNameCallback getProductsByProductNameCallback = null;

getProductsByNamePreparedStatementCreator = new GetProductsByNamePreparedStatementCreator(productName);

getProductsByProductNameCallback = new GetProductsByProductNameCallback();

productBos = jdbcTemplate.execute(getProductsByNamePreparedStatementCreator, getProductsByProductNameCallback);

return productBos;

}

private final class GetProductsByNamePreparedStatementCreator implements PreparedStatementCreator {

private String productName;

public GetProductByNamePreparedStatementCreator(String productName) {

this.productName = productName;

}

public PreparedStatement createPreparedStatement(Connection con) {

PreparedStatement pstmt = null;

pstmt = con.preparedStatement(SQL\_GET\_PRODUCTS\_BY\_NM);

pstmt.setString(1, productName);

return pstmt;

}

}

private final class GetProductsByProductNameCallback implements PreapredStatementCallback<List<ProductBo>> {

List<ProductBo> doInPreparedStatement(PreparedStatement pstmt) {

ResultSet rs = null;

ProductBo bo = null;

List<ProductBo> productBos = null;

productBos = new ArrayList();

rs = pstmt.executeQuery();

while(rs.next()) {

bo = new ProductBo();

bo.setProductNo(rs.getInt(1));

bo.setProductName(rs.getString(2));

bo.setDescription(rs.getString(3));

bo.setManufacturer(rs.getString(4));

bo.setPrice(rs.getFloat(5));

productBos.add(bo);

}

return productBos;

}

}

}

// spring JDBC API

class JdbcTemplate {

private DataSource dataSource;

public JdbcTemplate() {}

public JdbcTemplate(DataSource dataSource) {}

public void setDataSource(DataSource dataSource) {}

public Object execute(PreparedStatementCreator psc, PreparedStatementCallback pscb) {

Connection con = null;

PreparedStatement pstmt = null;

Object obj = null;

try {

con = dataSource.getConnection();

pstmt = psc.createPreparedStatement(con);

obj = pscb.doInPreparedStatement(pstmt);

// line

}catch(SqlException e) {

}finally {

try {

if(pstmt!= null) {

pstmt.close();

}

}catch(SqlException e) {

}

try {

if(con != null) {

con.close();

}

}catch(SqlException e) {}

}

return obj;

There are 2 ways we can execute a query while working with JDBC API.

1. Statement

2. PreparedStatement

What is the difference between Statement and PreparedStatement?

#1. Statement should be used for executing static SQL queries only and it is not recommended to use for executing dynamic SQL queries (which are supplied with values at runtime). Whereas PreparedStatement can be used for executing both static/dynamic SQL queries as well

If we use Statement for executing dynamic SQL queries, there is a possibility of SQL injection can take place, to avoid SQL injection we should use PreparedStatement only for executing dynamic SQL queries.

class EmployeeDao {

public List<Employee> getEmployee(String name) {

Connection con = null;

Statement stmt = null;

ResultSet rs = null;

try {

Class.forName(driverClassname);

con = DriverManager.getConnection(url, un, pwd);

stmt = con.createStatement();

rs = stmt.executeQuery("select \* from emp where emp\_nm like " + name);

while(rs.next()) {

// extract resultset and return

}

}catch(SqlException e) {

}finally {

// close resources

}

}

}

class EmployeeDao {

public List<Employee> getEmployee(String name) {

Connection con = null;

Statement stmt = null;

ResultSet rs = null;

try {

Class.forName(driverClassname);

con = DriverManager.getConnection(url, un, pwd);

pstmt = con.preparedStatement("select \* from emp where emp\_nm like ?");

pstmt.setString(1, name); // validates, is it query or value

while(rs.next()) {

// extract resultset and return

}

}catch(SqlException e) {

}finally {

// close resources

}

}

}

In the above code while substituting the values for the parameters defined in the query, the prepared statement validates whether the value is a "simple value" or another "query", if it looks like a query, it throws an exception indicating an sql attack

The morale is avoiding Statement and use PreparedStatement while executing dynamic SQL queries

#2.

If we are reusing the prepared statement object in repeatedly executing it, by substituting different set of values as an input while execution then only performance optimization takes place otherwise there is not big deal interms of performance between Statement and PreparedStatement

class EmployeeDao {

public List<Employee> findEmployeesByNames(List<String> names) {

Connection con = null;

Statement stmt = null;

ResultSet rs = null;

try {

Class.forName(driverClassname);

con = DriverManager.getConnection(url, un, pwd);

pstmt = con.preparedStatement("select \* from emp where emp\_nm like ?"); // query compilation takes place, and the query plan will be generated in the session

for(String name: names) {

pstmt.setString(1, name); // validates, is it query or value

while(rs.next()) {

// extract resultset and return

}

}

}catch(SqlException e) {

}finally {

// close resources

// close pstmt

// close con

}

return employees;

}

}

1. Statement

2. PreparedStatement

use Statement, if we are executing static sql queries then use statement, dont use Statement for dynamic sql queries, because it dont detect sql injection attack

If we are repeatedly executing a query over/against a set of values, then we can use PreparedStatement for optimizing the performance, rather than Statement

------------------------------------------------------------------------------------------

JdbcTemplate

has provided bunch of execute() methods to support working with callback mechanism in executing the sql queries and collecting the results

- T execute(PreparedStatementCreator, PreparedStatmentCallback<T>) // dynamic sql

insert, update, delete and select with where clause

- T execute(String, PreparedStatementCallback<T>) // static sql queries

- T execute(StatementCallback<T>) // static sql queries

interface StatementCallback<T> {

T doInStatement(Statement stmt) {

}

}

- T execute(CallableStatementCreator, CallableStatementCallback<T>) //pl/sql

- T execute(procedureName, CallableStatementCallback<T>) // pl/sql

- void execute(String sql) // ddl queries

- T execute(ConnectionCallback<T>)

interface ConnectionCallback<T> {

T doInConnection(Connection con) {

// give me connection

}

}

------------------------------------------------------------------------------------------

class ProductBo {

int productNo;

String productName;

String description;

float price;

// setters and getters

}

@Repository

class ProductDao {

private final String SQL\_GET\_PRODUCTS = "select product\_no, product\_nm, description, price from product";

@Autowired

private JdbcTemplate jdbcTemplate;

public ProductDao(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public List<ProductBo> getAllProducts() {

return jdbcTemplate.execute((new GetProductsStatementCallback()));

}

private final class GetProductsStatementCallback implements StatementCallback<List<ProductBo>> {

public List<ProductBo> doInStatement(Statement stmt) {

ProductBo bo = null;

ResultSet rs = null;

List<ProductBo> products = null;

rs = stmt.executeQuery(SQL\_GET\_PRODUCTS);

while(rs.next()) {

bo = new ProductBo();

// populate record into object

products.add(bo);

}

return products;

}

}

}

@Configuration

@PropertySource("classpath:db.properties")

@ComponentScan(basePackages={"com.ca.dao"})

class JavaConfig {

@Autowired

private Environment env;

@Bean

public DataSource dataSource() {

DriverManagerDataSource dataSource = null;

dataSource = new DriverManagerDataSource();

dataSource.setDriverClassname(env.getProperty("db.driverClassname"));

dataSource.setUrl(env.getProperty("db.url"));

dataSource.setUsername(env.getProperty("db.username"));

dataSource.setPassword(env.getProperty("db.password"));

return dataSource;

}

@Bean

public JdbcTemplate jdbcTemplate(DataSource dataSource) {

JdbcTemplate jdbcTemplate = null;

jdbcTemplate = new JdbcTemplate(dataSource);

return jdbcTemplate;

}

}

db.properties

-------------

db.driverClassname=com.mysql.cj.jdbc.Driver

db.url=jdbc:mysql://localhost:3306/hibdb

db.username=root

db.password=root

JdbcTemplate

1. T execute(PreparedStatementCreator, PreparedStatementCallback<T>) = dynamic sql queries

2. T execute(String sql, PreparedStatementCallback<T>) = static sql queries

3. T execute(StatementCallback<T>) = static sql queries

4. T execute(CallableStatementCreator, CallableStatementCallback<T>) = pl/sql procedures with parameters

5. T execute(String procedureName, CallableStatementCallback<T>) = pl/sql procedures without parameters

6. void execute(String sql) = ddl queries

7. T execute(ConnectionCallBack<T>) = if we want to execute multiple queries

------------------------------------------------------------------------------------------

class VehicleDao {

public int newVehicleForLease(VehicleBo vehicleBo, LeaseBo leaseBo) {

return jdbcTemplate.execute(new NewVehicleForLeaseConnectionCallback());

}

private final class NewVehicleForLeaseConnectionCallback implements ConnectionCallback<Integer> {

int doInConnection(Connection con) {

}

}

}

------------------------------------------------------------------------------------------

query-based approach

---------------------

How many vehicles are there in the vehicles table?

@Repository

class VehicleDao {

private final String SQL\_GET\_NO\_OF\_VEHICLES = "select count(1) from vehicles";

private final String SQL\_GET\_MAX\_LEASE\_AMOUNT = "SELECT max(lease\_amount) FROM vehicles";

private final String SQL\_GET\_MODELNM\_BY\_VEHICLE\_NO = "select model\_nm from vehicles where vehicle\_no = ?";

@Autowired

private JdbcTemplate jdbcTemplate;

public int getNoOfVehicles() {

return jdbcTemplate.queryForObject(SQL\_GET\_NO\_OF\_VEHICLES, Integer.class);

}

public int getMaxLeaseAmount() {

return jdbcTemplate.queryForObject(SQL\_GET\_MAX\_LEASE\_AMOUNT, Integer.class);

}

public String getModelName(int vehicleNo) {

return jdbcTemplate.queryForObject(SQL\_GET\_MODELNM\_BY\_VEHICLE\_NO, String.class, vehicleNo);

}

}

@Configuration

@PropertySource("classpath:db.properties")

@ComponentScan(basePackages = {"com.qa.dao"})

class JavaConfig {

@Autowired

private Environment env;

@Bean

public DataSource dataSource() {}

@Bean

public JdbcTemplate jdbcTemplate(DataSource dataSource) {}

}

class JdbcTemplate {

private DataSource dataSource;

public JdbcTemplate() {}

public JdbcTemplate(DataSource dataSource) {}

public T queryForObject(String sql, Class<T> classType) {

Connection con = null;

Statement stmt = null;

ResultSet rs = null;

Object ret = null;

try {

con = dataSource.getConnection();

stmt = con.createStatement();

rs = stmt.executeQuery(sql);

if(rs.next()) {

if(classType == Integer.class) {

return rs.getInt(1);

}else if(classType == Double.class) {

return rs.getDouble(1);

}else if(classType == String.class) {

return rs.getString(1);

}

}

}catch(SqlException e) {

throw new DataAccessException(e);

}finally {

// close rs, stmt, con

}

public T queryForObject(String sql, Class<T> classType, Object... params) {

Connection con = null;

PreparedStatement pstmt = null;

ResultSet rs = null;

Object ret = null;

int paramIndex = 1;

try {

con = dataSource.getConnection();

pstmt = con.preparedStatement(sql);

for(Object obj : params) {

if(obj instanceof Integer.class) {

pstmt.setInt(paramIndex, obj);

}else if(obj instanceof String.class) {

pstmt.setString(paramIndex, obj);

}else if(obj instanceof Float.class) {

pstmt.setFloat(paramIndex, obj);

}

paramIndex++;

}

rs = stmt.executeQuery(sql);

if(rs.next()) {

if(classType == Integer.class) {

return rs.getInt(1);

}else if(classType == Double.class) {

return rs.getDouble(1);

}else if(classType == String.class) {

return rs.getString(1);

}

}

}catch(SqlException e) {

throw new DataAccessException(e);

}finally {

// close rs, stmt, con

}

}

public void setDataSource(DataSource dataSource) {}

}

Query-Based Approach

----------------------

@Configuration

@PropertySource("classpath:db.properties")

@ComponentScan(basePackages = { "com.qa.dao" })

public class QAJavaConfig {

@Bean

public DataSource dataSource(@Value("${db.driverClassname}") String driverClassname, @Value("${db.url}") String url,

@Value("${db.username}") String username, @Value("${db.password}") String password) {

DriverManagerDataSource dataSource = null;

dataSource = new DriverManagerDataSource(url, username, password);

dataSource.setDriverClassName(driverClassname);

return dataSource;

}

@Bean

public JdbcTemplate jdbcTemplate(DataSource dataSource) {

return new JdbcTemplate(dataSource);

}

}

package com.qa.dao;

@Repository

public class VehicleDao {

private final String SQL\_NO\_OF\_VEHICLES = "select count(1) from vehicles";

private final String SQL\_MAX\_LEASE\_AMOUNT = "select max(lease\_amount) from vehicles";

private final String SQL\_GET\_MODEL\_NM\_BY\_VEHCILE\_NO = "select model\_nm from vehicles where vehicle\_no = ?";

private final String SQL\_GET\_VEHICLE\_BY\_VEHICLE\_NO="select vehicle\_no, model\_nm, manufacturer, color, price from vehicles where vehicle\_no = ?";

private final String SQL\_GET\_VEHICLES\_BY\_MODEL\_NM = "select vehicle\_no, model\_nm, manufacturer, color, lease\_amount from vehicles where model\_nm = ?";

private final String SQL\_GET\_VEHICLES\_BY\_MANUFACTURER = "select vehicle\_no, model\_nm, lease\_amount from vehicles where manufacturer like ?";

private final String SQL\_INSERT\_VEHICLE = "insert into vehicles(vehicle\_no, model\_nm, manufacturer, color, lease\_amount) values(?,?,?,?,?)";

@Autowired

private JdbcTemplate jdbcTemplate;

public int getNoOfVehicles() {

return jdbcTemplate.queryForObject(SQL\_NO\_OF\_VEHICLES, Integer.class);

}

public float maxLeaseAmount() {

return jdbcTemplate.queryForObject(SQL\_MAX\_LEASE\_AMOUNT, Float.class);

}

public String findModelName(int vehicleNo) {

return jdbcTemplate.queryForObject(SQL\_GET\_MODEL\_NM\_BY\_VEHCILE\_NO, String.class, vehicleNo);

}

public VehicleBo getVehicleDetails(int vehicleNo) {

return jdbcTemplate.queryForObject(SQL\_GET\_VEHICLE\_BY\_VEHICLE\_NO, new VehicleBoRowMapper(), vehicleNo);

}

public List<VehicleBo> getVehicles(String modelName) {

return jdbcTemplate.query(SQL\_GET\_VEHICLES\_BY\_MODEL\_NM, new VehicleBoRowMapper(), modelName);

}

public List<Map> getVehicles(String manufacturer) {

return jdbcTemplate.queryForList(SQL\_GET\_VEHICLES\_BY\_MANUFACTURER, manufacturer);

}

public int saveVehicle(VehicleBo bo) {

return jdbcTemplate.update(SQL\_INSERT\_VEHICLE, bo.getVehicleNo(), bo.getModelName(), bo.getManufactuter(), bo.getColor(), bo.getLeaseAmount());

}

// one record should be mapped to one object only

/\*\*

if our sql query is return a record of data, to covert the record into an object we need to write our own class implementing from RowMapper interface and write the logic for mapping record into object inside it and pass it as an input to JdbcTemplate

\*/

private final class VehicleBoRowMapper implements RowMapper<VehicleBo> {

public VehicleBo mapRow(ResultSet rs, int record) {

VehicleBo bo = new VehicleBo();

bo.setVehicleNo(rs.getInt(1));

bo.setModelName(rs.getString(2));

bo.setManufacturer(rs.getString(3));

bo.setColor(rs.getString(4));

bo.setLeaseAmount(rs.getFloat(5));

return bo;

}

}

}

class VehicleBo {

int vehicleNo;

String modelName;

String manufacturer;

String color;

float leaseAmount;

// accessors methods

}

interface RowMapper<T> {

T mapRow(ResultSet rs, int rowNumber);

}

#1. Connection

#2. PreparedStatement

#3. substitute the parameters

----------------------------------

pass sql & parameters

#4. execute the query

#5. iterate the resultset

#6. grab the record and store into VehicleBo object

#7. close the resources

class JdbcTempalte {

private DataSource dataSource;

public JdbcTemplate() {}

public JdbcTemplate(DataSource dataSource) {}

public T queryForObject(String sql, RowMapper<T> rowMapper, Object... params) {

T obj = null;

ResultSet rs = null;

Connection con = null;

PreparedStatement pstmt = null;

try {

con = dataSource.getConnection();

pstmt = con.preparedStatement(sql);

// substitute the parameters

rs = pstmt.executeQuery();

if(rs.next()) {

obj = rowMapper.mapRow(rs, 1);

}

}catch(SqlException e) {

throw new DataAccessException(e);

}finally {

// close resources

}

return obj;

}

public List<T> query(String sql, RowMapper<T> rowMapper, Object... params) {

int rowNum = 0;

T object = null;

ResultSet rs = null;

Connection con = null;

List<T> objects = null;

PreparedStatement pstmt = null;

try {

con = dataSource.getConnection();

pstmt = con.preparedStatement(sql);

// substitute parameters

rs = pstmt.executeQuery();

while(rs.next()) {

obj = rowMapper.mapRow(rs, rowNum);

objects.add(obj);

rowNum++;

}

}catch(SqlException e) {

throw new DataAccessException(e);

}finally {

// close resources

}

return objects;

}

public void setDataSource(DataSource dataSource) {}

}

select vehicle\_no, model\_nm, lease\_amount from vehicles where manufacturer like ?

vehicle\_no model\_nm lease\_amount

1 toyota corola 650

2 toyota corola 700

List<Map>

Map

vehicle\_no = 1

model\_nm = toyota corola

lease\_amount= 650

Map

vehicle\_no = 2

model\_nm = toyota corola

lease\_amount= 700

if your query is return partial columns which are generic then we cannot map the record into pojo class object, so each record should be mapped into a generic object type which is Map

So per each record one map and per list of records List<Map> objects are returned.

#1. How to work with autogenerated primary key columns of a table?

#2. How to implement pagination?

#3. How to work with rollup logic?

------------------------------------------------------------------------------------------

#1. working with autogenerated primary key columns

In the below tables, upon inserting the data into vehicles table, the database will generated vehicle\_no primary key column value, we need to fetch the vehicle\_no primary key column value that is generated by database to insert the data into vehicle\_details table

How to fetch the primarykey column value that is generated by the database after the insert operation?

vehicles

---------

vehicle\_no (pk) (AI) (surrogate key columns) (generated by the database)

model\_nm

manufacturer

color

registration\_no

fuel\_type

lease\_amount

java jdbc approach

--------------------

public int saveVehicle(VehicleBo bo) throws SqlException {

Connection con = null;

PreparedStatement pstmt = null;

try {

Class.forName("com.mysql.cj.jdbc.Driver");

con = DriverManager.getConnection(url, un, pwd);

pstmt = con.preparedStatement("insert into vehicles(model\_nm, manufacturer, color, registration\_no, fuel\_type, lease\_amount) values(?,?,?,?,?,?)", new String[]{"vehicle\_no"});

pstmt.setString(1, bo.getModelName());

pstmt.setString(2, bo.getManufacturer());

// substitute the positional parameters

pstmt.executeUpdate();

ResultSet rs = pstmt.getGeneratedKeys();

if(rs.next()) {

int vehicleNo = rs.getInt("vehicle\_no");

}

return vehicleNo;

}catch(SqlException e) {

e.printStackTrace();

throw e;

}finally {

// close resources

}

}

-----------------------------------------------------------------------------------------

@Configuration

class JavaConfig {...}

@Repository

class VehicleDao {

private final String SQL\_INSERT\_VEHICLE = "insert into vehicles(model\_nm, manufacturer, color, registration\_no, fuel\_type, lease\_amount) values(?,?,?,?,?,?)";

@Autowired

private JdbcTemplate jdbcTemplate;

public int saveVehicle(final VehicleBo bo) {

KeyHolder kh = new GeneratedKeyHolder();

jdbcTemplate.update(new PreparedStatementCreator() {

public PreparedStatment createPreparedStatement(Connection con) {

PreparedStatement pstmt = null;

pstmt = con.preparedStatement(SQL\_INSERT\_VEHICLE, new String[]{"vehicle\_no"});

pstmt.setString(1, bo.getModelName());

pstmt.setString(2, bo.getManufacturer());

pstmt.setString(3, bo.getColor());

pstmt.setString(4, bo.getRegistrationNo());

pstmt.setString(5, bo.getFuelType());

pstmt.setFloat(6, bo.getLeaseAmount());

return pstmt;

}

}, kh);

int vehicleNo = kh.get("vehicle\_no").intValue();

return vehicleNo;

}

}