----------Spring integration with JDBC----------

The standard way to access a database using java is to use a JDBC driver. The steps involved are creating or obtaining a connection from a connection pool, creating a statement, executing query or update statements and closing the connection. This looks simple, but you need to do this every time, the same steps over again in all methods that talk to the database,and that is not all, you also have to handle the SQL Exception almost everywhrere

SQLException – you always have to deal with SQLException when working with JDBC. The exception is thrown in almost all methods. But to get to the root of the problem is very difficult since SQLException does not tell us much about the problem (unless you catch the error code and write score of if-then statement).Even if you manage to get the problem out of SQL Exception, it will always be specific to a particular presistence mechanism, spring provides a hierarchy of Exceptions that cover almost all kinds of exceptions that a database can throw, Examples of these are **DataIntegrityViolationException**, **PermissionDeniedDataAccessException**,

Cannot AcquireLockException etc. The other good thing about this exception is that they need not be caught(unchecked exception) No matter what persistence mechanism you use, spring always throws this exception.

Templates: Most of the code for database access is repetitive, Opening connection, closing connection etc, Spring creates templates that handles most of this repetitve code, the developer ueses this template which does most of the work and just fills in the logic specific to the application. Developer does not have to worry about managing connections or exceptions, spring has template for various persistence framework such as plain JDBC, Hibernate,iBatis etc, Example of templates are **jdbc.core.jdbdTemplate**, **orm.hibernate** template etc.

DataSource- Spring requires the connection info to the persistence mechanism. This connection info is specified in the form of datasource, the datasource needs to be injected in the template, the datasource may also be created by the application server, the developer then uses JNDI to access the datasource.

J**DBC template** the jdbcTemplate class excutes 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.
Instances of the jdbdcTemplate class are **threadsafe** once configured, so you can configure a single instance of a jdbc template and then safely inject this shared reference into multiple DAOs.

RowMapper – RowMapper interface is uesed by the jdbcTemplate to map a resultSet row, the implementation of this interface maps a resultset row to a result object. The implementation does not have to worry about catching exceptions. Implementations must implements the method
T mapRow(ResultSet rs, int rowNum) throws SQLException

This is the JdbcTemplate method that uses it
public List query(String sql,RowMapper rowMapper)throws DataAccessException

Some Important jdbc Template methods:

Inserting a row into the table:

String sql =”INSERT INTO employee_table(employee_name, salary,email,gender) VALUES(?,?,?,?)”
int update = jdbcTemplate.update(sql, employee.getEmployeeName(), employee.getSalary(), employee.getEmail(),employee.getGender());

Querying and returning an object:

String sql= “SELECT * FROM employee_table WHERE employee_ id=?”
Employee employee = jdbcTemplate.queryForObject(sql,new Object[] {empId}, new EmployeeRowMapper());

Delete a row from the table;

int update =jdbcTemplate.update(“DELETE employee_table WHERE employee_id=?”, empId)

Updating a row into the table:

int update = jdbcTemplate.update(“UPDATE employee_table WHERE employee_id=?”, new Object[]{new Email, empId});

Querying and returning multiple objects

String sql =”SELECT * FROM employee_table”;
List<Employee> empList = jdbcTemplate.query(sql,new EmployeeRowMapper());

Downlaod jdbc driver (mySQl)

https://dev.mysql.com/downloads/file/?id=13597

DAO class----> have persistence logic for connect to data base performance percistence logic CRUD(insert,update.delete, retrive)
Service classs ---> service class have bussiness logic and validation

Connect database through jdbc in xml configuration file
1) required `org.springframework.jdbc.datasource.DriverManagerDataSource class to connect the database`

*2) give the **driverClassName** com.mysql.jdbc.Driver*
*3)url--->**jdbc:mysql://localhost:3306/test***
5)username
6)password

DriverManagerDataSource
Simple implementation of the standard JDBC `DataSource` interface, configuring the plain old JDBC `DriverManager` via bean properties, and returning a new `Connection` from every `getConnection` call.
NOTE: This class is not an actual connection pool; it does not actually pool Connections. It just serves as simple replacement for a full-blown connection pool, implementing the same standard interface, but creating new Connections on every call.

Useful for test or standalone environments outside of a Java EE container, either as a DataSource bean in a corresponding ApplicationContext or in conjunction with a simple JNDI environment. Pool-assuming `Connection.close()` calls will simply close the Connection, so any DataSource-aware persistence code should work.

NOTE: Within special class loading environments such as OSGi, this class is effectively superseded by `SimpleDriverDataSource` due to general class loading issues with the JDBC DriverManager that be resolved through direct Driver usage (which is exactly what SimpleDriverDataSource does).

----------Example----------

XML file

```
<bean id="dataSource"
            class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <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="123" />
      </bean>

      <bean id="employeeService"
class="com.infotech.service.EmployeeServiceImpl">
      <property name="employeeDAO" ref="employeeDAO"/>
      </bean>

      <bean id="employeeDAO" class="com.infotech.dao.EmployeeDAOImpl">
         <property name="dataSource" ref="dataSource"/>

      </bean>
```

----------DAO----------

```
public interface EmployeeDAO {


	public  abstract void createEmployee(Employee employee);
	public  abstract Employee getEmployee(int empId);
	public abstract void updateEmployeeEmailById(String email,int emplId);
	public abstract void deleteEmployeeById(int empId);
	public abstract List<Employee> getAllEmployess();
}

public class EmployeeDAOImpl implements EmployeeDAO{

	private DataSource dataSource;

	public void setDataSource(DataSource dataSource) {
		this.dataSource = dataSource;
	}

	@Override
	public void createEmployee(Employee employee) {

		Connection connection =null;
		PreparedStatement ps =null;
		try {
			String SQL = "INSERT INTO employee_table (employee_name,
salary,email, gender) VALUES(?,?,?,?)";
			connection = dataSource.getConnection();
			ps = connection.prepareStatement(SQL);

			ps.setString(1,employee.getEmployeeName());
			ps.setDouble(2,employee.getSalary());
			ps.setString(3,employee.getEmail());
			ps.setString(4,employee.getGender());
		int result =	ps.executeUpdate();

		System.out.println("count the number is effected "+result);


		}catch(Exception e) {
			e.printStackTrace();
		}
		finally{
			if(connection!=null) {

				try {
```

```
                        connection.close();
                    } catch (SQLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            }

    }

    @Override
    public Employee getEmployee(int empId) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void updateEmployeeEmailById(String email, int emplId) {
        // TODO Auto-generated method stub

    }

    @Override
    public void deleteEmployeeById(int empId) {
        // TODO Auto-generated method stub

    }

    @Override
    public List<Employee> getAllEmployess() {
        // TODO Auto-generated method stub
        return null;
    }

}
```

----------Service----------

```
public interface EmployeeService {

    public  abstract void addEmployee(Employee employee);
    public  abstract Employee fetchEmployee(int empId);
    public abstract void updateEmployeeEmailById(String email,int emplId);
    public abstract void deleteEmployeeById(int empId);
    public abstract List<Employee> fetchAllEmployee();

}


public class EmployeeServiceImpl implements EmployeeService{


    private EmployeeDAO employeeDAO;


    public void setEmployeeDAO(EmployeeDAO employeeDAO) {
        this.employeeDAO = employeeDAO;
    }
```

```
	@Override
	public void addEmployee(Employee employee) {

		employeeDAO.createEmployee(employee);

	}

	@Override
	public Employee fetchEmployee(int empId) {

		return employeeDAO.getEmployee(empId);
	}

	@Override
	public void updateEmployeeEmailById(String email, int emplId) {

		employeeDAO.updateEmployeeEmailById(email, emplId);

	}

	@Override
	public void deleteEmployeeById(int empId) {

		employeeDAO.deleteEmployeeById(empId);

	}

	@Override
	public List<Employee> fetchAllEmployee() {

		return employeeDAO.getAllEmployess();
	}

}
```

------------------------------Main class--------------------------

```
public class ClientTest {

	public static void main(String args[]) {

		AbstractApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");


		EmployeeService employeeService =
context.getBean("employeeService",EmployeeService.class);

		Employee employee = new Employee();
		employee.setEmployeeName("johan");
		employee.setEmail("johan@gmailc.om");
		employee.setSalary(25000);
		employee.setGender("MALE");

		employeeService.addEmployee(employee);
```

```
            context.close();

      }

}
```

```
------------------Full CRUD operations(jdbcTemplate)-----------------------
```

```
-------------------------DAO-------------------------------
public interface EmployeeDAO {

      public  abstract void createEmployee(Employee employee);
      public  abstract Employee getEmployee(int empId);
      public abstract int updateEmployeeEmailById(String email,int emplId);
      public abstract int deleteEmployeeById(int empId);
      public abstract List<Employee> getAllEmployess();



}

public class EmployeeDAOImpl implements EmployeeDAO{


      private DataSource dataSource;

      private JdbcTemplate jdbcTemplate;



      public void setDataSource(DataSource dataSource) {
            this.dataSource = dataSource;
            jdbcTemplate = new JdbcTemplate(this.dataSource);
      }

      @Override
      public void createEmployee(Employee employee) {

            String SQL = "INSERT INTO employee_table (employee_name,
salary,email, gender) VALUES(?,?,?,?)";
```

```
	int result =	jdbcTemplate.update(SQL, new Object[]
{employee.getEmployeeName(),employee.getSalary(),employee.getEmail(),employee.ge
tGender()});

	}

	@Override
	public Employee getEmployee(int empId) {
		String SQL = "SELECT * FROM employee_table where employee_id =?";
		Employee employee = jdbcTemplate.queryForObject(SQL, new
EmployeeRowMapper(),empId);
		return employee;
	}

	@Override
	public int updateEmployeeEmailById(String email, int emplId) {

		String SQL = "UPDATE employee_table set email=?  where
employee_id=?";
		int result = jdbcTemplate.update(SQL, email,emplId);
		return result;


	}

	@Override
	public int deleteEmployeeById(int empId) {

		String SQL = "DELETE from employee_table where employee_id=?";
		return jdbcTemplate.update(SQL, empId);


	}

	@Override
	public List<Employee> getAllEmployess() {
		String SQL = "SELECT * FROM employee_table";
		return jdbcTemplate.query(SQL, new EmployeeRowMapper());
	}

}


public class EmployeeRowMapper implements RowMapper<Employee> {

	@Override
	public Employee mapRow(ResultSet rs, int arg1) throws SQLException {

		Employee employee = new Employee();
		employee.setEmployeeId(rs.getInt(1));
		employee.setEmployeeName(rs.getString(2));
		employee.setSalary(rs.getDouble(3));
		employee.setEmail(rs.getString(4));
		employee.setGender(rs.getString(5));
		return employee;
	}
```

```
}


------------------------Service------------------------------

public interface EmployeeService {

	public  abstract void addEmployee(Employee employee);
	public  abstract Employee fetchEmployeeById(int empId);
	public abstract int updateEmployeeEmailById(String email,int emplId);
	public abstract int deleteEmployeeById(int empId);
	public abstract List<Employee> fetchAllEmployee();

}

public class EmployeeServiceImpl implements EmployeeService{


	private EmployeeDAO employeeDAO;



	/*public EmployeeDAO getEmployeeDAO() {
		return employeeDAO;
	}*/

	public void setEmployeeDAO(EmployeeDAO employeeDAO) {
		this.employeeDAO = employeeDAO;
	}

	@Override
	public void addEmployee(Employee employee) {

		employeeDAO.createEmployee(employee);

	}

	@Override
	public Employee fetchEmployeeById(int empId) {

		return employeeDAO.getEmployee(empId);
	}

	@Override
	public int updateEmployeeEmailById(String email, int emplId) {

		return employeeDAO.updateEmployeeEmailById(email, emplId);

	}

	@Override
	public int deleteEmployeeById(int empId) {

		return employeeDAO.deleteEmployeeById(empId);

	}
```

```
	@Override
	public List<Employee> fetchAllEmployee() {

		return employeeDAO.getAllEmployess();
	}

}public class ClientTest {

	public static void main(String args[]) {

		AbstractApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");


		EmployeeService employeeService =
context.getBean("employeeService",EmployeeService.class);

		Employee employee = new Employee();
		employee.setEmployeeName("mohan");
		employee.setEmail("mohan@gmailc.om");
		employee.setSalary(400000);
		employee.setGender("MALE");

		//employeeService.addEmployee(employee);



		int result = employeeService.updateEmployeeEmailById("jk@gmail.com",
1);

		System.out.println("updated record is "+result);


		Employee emp = employeeService.fetchEmployeeById(5);

		System.out.println("------------specific object------------------");

		System.out.println(emp);

		List<Employee> listEmployee = employeeService.fetchAllEmployee();

		System.out.println("----------------list of
employee-----------------");

		for(Employee empl : listEmployee) {


			System.out.println(empl);

		}
		int delete = employeeService.deleteEmployeeById(5);
		System.out.println("delete record-------"+delete);

		context.close();


	}
```

```
}
```

```
----------------------------main class-------------------
```

```
public class ClientTest {

	public static void main(String args[]) {

		AbstractApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");


		EmployeeService employeeService =
context.getBean("employeeService",EmployeeService.class);

		Employee employee = new Employee();
		employee.setEmployeeName("mohan");
		employee.setEmail("mohan@gmailc.om");
		employee.setSalary(400000);
		employee.setGender("MALE");

		//employeeService.addEmployee(employee);



		int result = employeeService.updateEmployeeEmailById("jk@gmail.com",
1);

		System.out.println("updated record is "+result);


		Employee emp = employeeService.fetchEmployeeById(5);

		System.out.println("------------specific object------------------");

		System.out.println(emp);

		List<Employee> listEmployee = employeeService.fetchAllEmployee();

		System.out.println("-------list of employee----------------");

		for(Employee empl : listEmployee) {


			System.out.println(empl);

		}
		int delete = employeeService.deleteEmployeeById(5);
		System.out.println("delete record-------"+delete);

		context.close();


	}

}
```

```
<bean id="dataSource"
            class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <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="123" />
      </bean>

      <bean id="employeeService"
class="com.infotech.service.EmployeeServiceImpl">
      <property name="employeeDAO" ref="employeeDAO"/>
      </bean>

      <bean id="employeeDAO" class="com.infotech.dao.EmployeeDAOImpl">
         <property name="dataSource" ref="dataSource"/>
      </bean>
```

use property file to store database driver, url, username,passowrd

base.properties

```
db.driver = com.mysql.jdbc.Driver
db.url = jdbc:mysql://localhost:3306/test
db.username = root
db.password = 123
```

-------------------------xml file-----------------------

```
      <bean id="dataSource"
            class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <property name="driverClassName" value="${db.driver}" />
            <property name="url" value="${db.url}"/>
            <property name="username" value="${db.username}" />
            <property name="password" value="${db.password}" />
      </bean>

      <bean id="employeeService"
class="com.infotech.service.EmployeeServiceImpl">
      <property name="employeeDAO" ref="employeeDAO"/>
      </bean>

      <bean id="employeeDAO" class="com.infotech.dao.EmployeeDAOImpl">
         <property name="dataSource" ref="dataSource"/>
      </bean>

      <!-- to read properties file -->
      <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
        <list>
        <value>database.properties</value>
        </list>

        </property>
      </bean>
```

------------required to read properties file in xml ------------

```
      <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
        <list>
        <value>database.properties</value>
        </list>

        </property>
      </bean>


other code is same a previous
```

```
---------------- previous example Code refectores --------------------
```

creating object with new keyword in spring framework is not good way so we can avoid to create object ,
so will be used dependency injection xml file confuration means we can just pass the class name and property then container automatically create the object.

Problem in code

```
public class EmployeeDAOImpl implements EmployeeDAO{


      private DataSource dataSource;

      private JdbcTemplate jdbcTemplate;



      public void setDataSource(DataSource dataSource) {
            this.dataSource = dataSource;

//it is not standard to create object manually
            jdbcTemplate = new JdbcTemplate(this.dataSource);
      }
```

Solutions:

change in XML file and EmployeeDAOImpl logic

```
<bean id="employeeDAO" class="com.infotech.dao.EmployeeDAOImpl">
        <property name="jdbcTemplate" ref="jdbcTemplate"/>
      </bean>

<!-- JdbcTemplate class have constructor to take DataSoruce type so we can use
constructor injection in JdbcTemplate of DataSource -->
```

```
	<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">

	<constructor-arg name="dataSource" ref="dataSource"></constructor-arg>
	</bean>
```

------------------full xml file----------------

```
<bean id="employeeService" class="com.infotech.service.EmployeeServiceImpl">
	<property name="employeeDAO" ref="employeeDAO"/>
	</bean>

	<bean id="employeeDAO" class="com.infotech.dao.EmployeeDAOImpl">
	   <property name="jdbcTemplate" ref="jdbcTemplate"/>
	</bean>

	<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">

	<constructor-arg name="dataSource" ref="dataSource"></constructor-arg>
	</bean>

	<bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
	  <property name="locations">
	  <list>
	  <value>database.properties</value>
	  </list>

	  </property>
	</bean>
```

----------DAO class----------

```
public class EmployeeDAOImpl implements EmployeeDAO{

	private JdbcTemplate jdbcTemplate;

	public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}
```

```
//other are same as previous example

}
```

other programe logic are same as previous example

--------------Annotation based JDBC-------------------------

@Service –annotaed a service class as @Service it have bussiness logic

@Repository-->Repository annotation basically used by DAO class which have logic of access the database data

@Autowired–autowired annotation we can make dependecy class as autowired which is created that object

to serach our bean and enable annotatio we have to configure in xml

```
<context:component-scan base-package="com.infotech.dao"></context:component-scan>
	<context:component-scan base-package="com.infotech.service"></context:component-scan>
```

configure the package name in xml for searching the bean and create the object of bean by spring container

```
base-package="com.infotech.dao"

----------------------XML ----------------------------------

<bean id="dataSource"
		class="org.springframework.jdbc.datasource.DriverManagerDataSource">
		<property name="driverClassName" value="${db.driver}" />
		<property name="url" value="${db.url}"/>
		<property name="username" value="${db.username}" />
		<property name="password" value="${db.password}" />
	</bean>

	<context:component-scan base-package="com.infotech.dao"></context:component-scan>
```

```
	<context:component-scan base-
package="com.infotech.service"></context:component-scan>

	<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">

	<constructor-arg name="dataSource" ref="dataSource"></constructor-arg>
	</bean>

	<bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
	  <property name="locations">
	  <list>
	  <value>database.properties</value>
	  </list>

	  </property>
	</bean>

</beans>

------------------------------DAO--------------------------------------

public interface EmployeeDAO {

	public  abstract void createEmployee(Employee employee);
	public  abstract Employee getEmployee(int empId);
	public abstract int updateEmployeeEmailById(String email,int emplId);
	public abstract int deleteEmployeeById(int empId);
	public abstract List<Employee> getAllEmployess();


	}

@Repository
public class EmployeeDAOImpl implements EmployeeDAO{

	@Autowired
	private JdbcTemplate jdbcTemplate;

	public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	@Override
	public void createEmployee(Employee employee) {

		String SQL = "INSERT INTO employee_table (employee_name,
salary,email, gender) VALUES(?,?,?,?)";

	int result =	jdbcTemplate.update(SQL, new Object[]
{employee.getEmployeeName(),employee.getSalary(),employee.getEmail(),employee.ge
tGender()});

	}

	@Override
	public Employee getEmployee(int empId) {
```

```
            String SQL = "SELECT * FROM employee_table where employee_id =?";
            Employee employee = jdbcTemplate.queryForObject(SQL, new
EmployeeRowMapper(),empId);
            return employee;
      }

      @Override
      public int updateEmployeeEmailById(String email, int emplId) {

            String SQL = "UPDATE employee_table set email=?  where
employee_id=?";
            int result = jdbcTemplate.update(SQL, email,emplId);
            return result;


      }

      @Override
      public int deleteEmployeeById(int empId) {

            String SQL = "DELETE from employee_table where employee_id=?";
            return jdbcTemplate.update(SQL, empId);


      }

      @Override
      public List<Employee> getAllEmployess() {
            String SQL = "SELECT * FROM employee_table";
            return jdbcTemplate.query(SQL, new EmployeeRowMapper());
      }

}


public class EmployeeRowMapper implements RowMapper<Employee> {

      @Override
      public Employee mapRow(ResultSet rs, int arg1) throws SQLException {

            Employee employee = new Employee();
            employee.setEmployeeId(rs.getInt(1));
            employee.setEmployeeName(rs.getString(2));
            employee.setSalary(rs.getDouble(3));
            employee.setEmail(rs.getString(4));
            employee.setGender(rs.getString(5));
            return employee;
      }


}
-----------------------Service-----------------------------------

public interface EmployeeService {

      public  abstract void addEmployee(Employee employee);
      public  abstract Employee fetchEmployeeById(int empId);
      public abstract int updateEmployeeEmailById(String email,int emplId);
```

```
	public abstract int deleteEmployeeById(int empId);
	public abstract List<Employee> fetchAllEmployee();

}

@Service("employeeService")
public class EmployeeServiceImpl implements EmployeeService{


	@Autowired
	private EmployeeDAO employeeDAO;



	/*public EmployeeDAO getEmployeeDAO() {
		return employeeDAO;
	}*/

	public void setEmployeeDAO(EmployeeDAO employeeDAO) {
		this.employeeDAO = employeeDAO;
	}

	@Override
	public void addEmployee(Employee employee) {

		employeeDAO.createEmployee(employee);

	}

	@Override
	public Employee fetchEmployeeById(int empId) {

		return employeeDAO.getEmployee(empId);
	}

	@Override
	public int updateEmployeeEmailById(String email, int emplId) {

		return employeeDAO.updateEmployeeEmailById(email, emplId);

	}

	@Override
	public int deleteEmployeeById(int empId) {

		return employeeDAO.deleteEmployeeById(empId);

	}

	@Override
	public List<Employee> fetchAllEmployee() {

		return employeeDAO.getAllEmployess();
	}

}

----------------------------Main class---------------------------
```

```
public class ClientTest {

	public static void main(String args[]) {

		AbstractApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");


		EmployeeService employeeService =
context.getBean("employeeService",EmployeeService.class);

		Employee employee = new Employee();
		employee.setEmployeeName("mohan1");
		employee.setEmail("mohan@gmailc1.om");
		employee.setSalary(40000);
		employee.setGender("MALE");

		employeeService.addEmployee(employee);

		Employee emp = employeeService.fetchEmployeeById(1);

		System.out.println("------------specific object-----------------");

		System.out.println(emp);

		List<Employee> listEmployee = employeeService.fetchAllEmployee();

		System.out.println("----------------list of
employee-----------------");

		for(Employee empl : listEmployee) {


			System.out.println(empl);

		}


		context.close();


	}

}
```

----------DAO support----------

Spring framework provides excellent support to **JDBC**, it provides a super powerful utility class called **"JdbcTemplate"**which helps us avoid boiler-plate code from our database operations such as Creating Connection, Statement, Closing the Resultset and Connection, Exception handling, Transaction management etc. In this **Spring JdbcTemplate** Example, let's understand how **JdbcTemplate** eases the development effort.
It internal use Jdbc template just we call getJdbcTemplate();

The **Data Access Object (DAO) support** in Spring is aimed at making it easy to work with data access technologies like**JDBC**, **Hibernate** or **JDO** in a consistent way. This allows one to switch between the aforementioned persistence technologies fairly easily and it also allows one to code without worrying about catching exceptions that are specific to each technology.

• **JdbcDaoSupport** – superclass for JDBC data access objects. Requires a DataSource to be provided; in turn, this class provides a **JdbcTemplate** instance initialized from the supplied DataSource to subclasses.
• **HibernateDaoSupport** – superclass for Hibernate data access objects. Requires a SessionFactory to be provided; in turn, this class provides a HibernateTemplate instance initialized from the supplied SessionFactory to subclasses. Can alternatively be initialized directly via a HibernateTemplate, to reuse the latters settings like SessionFactory, flush mode, exception translator, and so forth.
• **JdoDaoSupport** – super class for JDO data access objects. Requires a PersistenceManagerFactory to be provided; in turn, this class provides a JdoTemplate instance initialized from the supplied PersistenceManagerFactory to subclasses.
• **JpaDaoSupport** – super class for JPA data access objects. Requires a EntityManagerFactory to be provided; in turn, this class provides a JpaTemplate

In Spring JDBC Framework there are many DAO support classes which help to reduce the configuration of JdbcTemplate, SimpleJdbcTemplate and NamedParamJdbcTemplate with dataSource object.

Advantage
Reduce the boiler pliate problem we don't need to create JdbcTemplate object explicity it internaly create JdbcTemplate class object we just access that object
Disadvantege
inheritance problem when we can extends JdbvDao class then we can not extends another class because java doen't support multple inheritance

Example

```
public class EmployeeDAOImpl extends JdbcDaoSupport implements EmployeeDAO{



	@Override
	public void createEmployee(Employee employee) {

		String SQL = "INSERT INTO employee_table (employee_name,
salary,email, gender) VALUES(?,?,?,?)";

	int result =	getJdbcTemplate().update(SQL, new Object[]
{employee.getEmployeeName(),employee.getSalary(),employee.getEmail(),employee.ge
tGender()});

	}
```

```
	@Override
	public Employee getEmployee(int empId) {
		String SQL = "SELECT * FROM employee_table where employee_id =?";
		Employee employee = getJdbcTemplate().queryForObject(SQL, new
EmployeeRowMapper(),empId);
		return employee;
	}

	@Override
	public int updateEmployeeEmailById(String email, int emplId) {

		String SQL = "UPDATE employee_table set email=?  where
employee_id=?";
		int result = getJdbcTemplate().update(SQL, email,emplId);
		return result;


	}

	@Override
	public int deleteEmployeeById(int empId) {

		String SQL = "DELETE from employee_table where employee_id=?";
		return getJdbcTemplate().update(SQL, empId);


	}

	@Override
	public List<Employee> getAllEmployess() {
		String SQL = "SELECT * FROM employee_table";
		return getJdbcTemplate().query(SQL, new EmployeeRowMapper());
	}

}
```

Note other class is same a previous example

----------Spring NamedParameterJdbcTemplate----------

In JdbcTemplate, SQL parameters are represented by a special placeholder "**?**" symbol and bind it by position. The problem is whenever the order of parameter is changed, you have to change the parameters bindings as well, it's error prone and cumbersome to maintain it.

To fix it, you can use "**Named Parameter**", whereas SQL parameters are defined by a starting colon follow by a name, rather than by position. In additional, the named parameters are only support in **SimpleJdbcTemplate** and **NamedParameterJdbcTemplate**.

In NamedParameter Query we can use Map or `MapSqlParameterSource` class to set key and value pair query

----------XML----------

```
<bean id="employeeService" class="com.infotech.service.EmployeeServiceImpl">
```

```
<property name="employeeDAO" ref="employeeDAO"></property>
</bean>
<bean id="employeeDAO" class="com.infotech.dao.EmployeeDAOImpl">

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


</bean>
<bean id="namedParameterJdbcTemplate"
class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">

<constructor-arg name="dataSource" ref="dataSource"></constructor-arg>
</bean>


      <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
        <list>
        <value>database.properties</value>
        </list>

        </property>
      </bean>

</beans>
```

```
public class EmployeeDAOImpl implements EmployeeDAO{


      private NamedParameterJdbcTemplate namedParameterJdbcTemplate;



      public void setNamedParameterJdbcTemplate(NamedParameterJdbcTemplate
namedParameterJdbcTemplate) {
            this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;
      }

      @Override
      public void createEmployee(Employee employee) {

            String SQL = "INSERT INTO employee_table (employee_name,
salary,email, gender) VALUES(:name,:salary,:email,:gender)";
            MapSqlParameterSource source = new MapSqlParameterSource();
            source.addValue("name", employee.getEmployeeName());
            source.addValue("salary", employee.getSalary());
```

```
            source.addValue("email", employee.getEmail());
            source.addValue("gender", employee.getGender());

      int result =      namedParameterJdbcTemplate.update(SQL, source);
      System.out.println("create result "+result);

      }

      @Override
      public Employee getEmployee(int empId) {
            String SQL = "SELECT * FROM employee_table where employee_id =
:empId";

            Map<String,Object>map = new HashMap<String,Object>();
            map.put("empId", empId);
            Employee employee =
namedParameterJdbcTemplate.queryForObject(SQL,map, new EmployeeRowMapper());
            return employee;
      }

      @Override
      public int updateEmployeeEmailById(String email, int emplId) {

            String SQL = "UPDATE employee_table set email=:email  where
employee_id=:empId";
            MapSqlParameterSource source = new MapSqlParameterSource();
            source.addValue("email", email);
            source.addValue("empId", emplId);
            int result = namedParameterJdbcTemplate.update(SQL, source);
            return result;


      }

      @Override
      public int deleteEmployeeById(int empId) {

            String SQL = "DELETE from employee_table where employee_id=?";

            MapSqlParameterSource source = new MapSqlParameterSource();
            source.addValue("empId", empId);
            int result = namedParameterJdbcTemplate.update(SQL, source);

            return result;


      }

      @Override
      public List<Employee> getAllEmployess() {

            String SQL = "SELECT * FROM employee_table";
            return namedParameterJdbcTemplate.query(SQL, new
EmployeeRowMapper());

      }

}
```

Note: NOther are same as previous example only these class logic is changed

----------extends `NamedParameterJdbcDaoSupport`----------

```
public class EmployeeDAOImpl extends NamedParameterJdbcDaoSupport  implements EmployeeDAO{



	@Override
	public void createEmployee(Employee employee) {

		String SQL = "INSERT INTO employee_table (employee_name, salary,email, gender) VALUES(:name,:salary,:email,:gender)";
		MapSqlParameterSource source = new MapSqlParameterSource();
		source.addValue("name", employee.getEmployeeName());
		source.addValue("salary", employee.getSalary());
		source.addValue("email", employee.getEmail());
		source.addValue("gender", employee.getGender());

	int result =	getNamedParameterJdbcTemplate().update(SQL, source);
	System.out.println("create result "+result);

	}

	@Override
	public Employee getEmployee(int empId) {
		String SQL = "SELECT * FROM employee_table where employee_id = :empId";

		Map<String,Object>map = new HashMap<String,Object>();
		map.put("empId", empId);
		Employee employee = getNamedParameterJdbcTemplate().queryForObject(SQL,map, new EmployeeRowMapper());
		return employee;
	}

	@Override
	public int updateEmployeeEmailById(String email, int emplId) {

		String SQL = "UPDATE employee_table set email=:email  where employee_id=:empId";
		MapSqlParameterSource source = new MapSqlParameterSource();
		source.addValue("email", email);
		source.addValue("empId", emplId);
		int result = getNamedParameterJdbcTemplate().update(SQL, source);
		return result;


	}
```

```
	@Override
	public int deleteEmployeeById(int empId) {

		String SQL = "DELETE from employee_table where employee_id=?";

		MapSqlParameterSource source = new MapSqlParameterSource();
		source.addValue("empId", empId);
		int result = getNamedParameterJdbcTemplate().update(SQL, source);

		return result;


	}

	@Override
	public List<Employee> getAllEmployess() {

		String SQL = "SELECT * FROM employee_table";
		return getNamedParameterJdbcTemplate().query(SQL, new
EmployeeRowMapper());

	}

}
```

only these bean are changed and other logic and beans are same as previous

The benefits of using stored procedures in SQL Server rather than application code stored locally on client computers include:

1. They allow modular programming.
2. They allow faster execution.
3. They can reduce network traffic.
4. They can be used as a security mechanism.

----------Store procedure in jdbc----------

Intsall my sql-workbrench for linex UI

sudo apt-get install mysql-workbench

create store procedure in mysql workbrench

```
CREATE DEFINER=`root`@`localhost` PROCEDURE
`getEmployeeNameAndSalaryById`(IN emp_id INT,OUT emp_sal DOUBLE,OUT
emp_name VARCHAR(100))
BEGIN

SELECT employee_name, salary INTO emp_name, emp_sal FROM employee_table
WHERE employee_id = emp_id;
END;
```

execute store procedure

```
set @emp_id=2;
CALL
`test`.`getEmployeeNameAndSalaryById`(@emp_id,@emp_sal,@emp_name);
select @emp_sal,@emp_name;
```

---

how to call store procedure in Programer approach

1) first we have to create SimpleJdbcCall object that have method to call store procedure in DAO class

```
example.private SimpleJdbcCall simpleJdbcCall;
```

2) register the store procedure in SimpleJdbcCall

```
      simpleJdbcCall.withProcedureName("getEmployeeNameAndSalaryById");
```

3) create MapSqlParameterSource or map object to pass as input as key value paire to store procedure (input key name must be same as store procedure in parameter name)

```
MapSqlParameterSource inputMap = new MapSqlParameterSource();
            inputMap.addValue("emp_id", empId);
```

4)to execute the storedprocedure call the execute() method and pass the map obj. And it return type is map type

```
Map<String,Object> outMap= simpleJdbcCall.execute(inputMap);
```

5) create sperate each out put value from stored procedure and convert into object type (we get the value from stored procedure as key value paire)so we have must pass the key same as out parameter of stored procedure

```
Employee employee = new Employee();
employee.setEmployeeName((String)outMap.get("emp_name"));//key name must same as
store procedure key
employee.setSalary((double)outMap.get("emp_sal"));
```

Example

----------DAO class----------

```
public interface EmployeeDAO {

      public abstract Employee getEmployeeNameAndSalaryById(int empId);

}
```

```
public class EmployeeDAOImpl implements EmployeeDAO{

	private SimpleJdbcCall simpleJdbcCall;

	public void setSimpleJdbcCall(SimpleJdbcCall simpleJdbcCall) {
		this.simpleJdbcCall = simpleJdbcCall;
	}



	@Override
	public Employee getEmployeeNameAndSalaryById(int empId) {
		// Registered the store procedure name in simpleJdbcTemplate
		simpleJdbcCall.withProcedureName("getEmployeeNameAndSalaryById");

		MapSqlParameterSource inputMap = new MapSqlParameterSource();
		inputMap.addValue("emp_id", empId);//key name must be same as store
procedure input key name


		//execute the stored procedure
		Map<String,Object> outMap= simpleJdbcCall.execute(inputMap);

		Employee employee = new Employee();
		employee.setEmployeeName((String)outMap.get("emp_name"));//key name
must same as store procedure key
		employee.setSalary((double)outMap.get("emp_sal"));


		return employee;
	}

}
```

```
------------------Modal---------------------------------

public class Employee {


	private int employeeId;
	private String employeeName;
	private double salary;
	private String email;
	private String gender;

//setter and getter method

}
```

```
--------------------------Service class-----------------------

public interface EmployeeService {

	public abstract Employee getEmployeeNameAndSalaryById(int empId);

}
```

```
public class EmployeeServiceImpl implements EmployeeService{

	private EmployeeDAO employeeDAO;

	public void setEmployeeDAO(EmployeeDAO employeeDAO) {
		this.employeeDAO = employeeDAO;
	}



	@Override
	public Employee getEmployeeNameAndSalaryById(int empId) {
		// TODO Auto-generated method stub
		return employeeDAO.getEmployeeNameAndSalaryById(empId);
	}

}
```

```
------------------main class-----------------------------
```

```
public static void main(String[] args) {
		// TODO Auto-generated method stub

		AbstractApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

		EmployeeService employeeService =
context.getBean("employeeService",EmployeeService.class);

		Employee employee = employeeService.getEmployeeNameAndSalaryById(2);

		System.out.println("emp name--- "+employee.getEmployeeName() +"
salary--- "+employee.getSalary());

		context.close();
	}
```

```
---------------------------XML---------------------------------
```

```
<bean id="dataSource"
		class="org.springframework.jdbc.datasource.DriverManagerDataSource">
		<property name="driverClassName" value="${db.driver}" />
		<property name="url" value="${db.url}"/>
		<property name="username" value="${db.username}" />
		<property name="password" value="${db.password}" />
	</bean>


<bean id="employeeService" class="com.infotech.service.EmployeeServiceImpl">
<property name="employeeDAO" ref="employeeDAO"></property>
</bean>

<bean id="employeeDAO" class="com.infotech.dao.EmployeeDAOImpl">

<property name="simpleJdbcCall" ref="simpleJdbcCall"></property>
</bean>

<bean id="simpleJdbcCall"
class="org.springframework.jdbc.core.simple.SimpleJdbcCall">
```

```
<constructor-arg name="dataSource" ref="dataSource"></constructor-arg>
</bean>


      <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
        <list>
        <value>database.properties</value>
        </list>

        </property>
      </bean>
```

------------------Spring with hibernate--------------------

download jar file

https://sourceforge.net/projects/hibernate/files/hibernate-orm/5.2.12.Final/hibernate-release-5.2.12.Final.zip/download