1. reduce boilerplate code
2. configuration
3. jdbcTemplate

step 1: create DAO and DTO and database tables

```
CREATE TABLE account (
  id int not null primary key,
  name VARCHAR(20) NOT NULL,
  balance double NOT NULL
);

public class Account {
	private int id;
	private String name;
	private double balance;
}


public interface AccountDao {
	public void update(Account account);
	public void save(Account account);
	public Account find(int id);
}

public class AccountDaoImp implements AccountDao
{
	private DataSource dataSource;

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

Now implementation of the method:

```
		String sql = "INSERT INTO account (id, name, balance ) VALUES (?, ?, ?)";
		Connection conn = null;
```

```
                try {
                        conn = dataSource.getConnection();
                        PreparedStatement ps = conn.prepareStatement(sql);
                        ps.setInt(1, account.getId());
                        ps.setString(2,account.getName() );
                        ps.setDouble(3, account.getBalance());
                        ps.executeUpdate();
                        ps.close();

                } catch (SQLException e) {
                        throw new RuntimeException(e);

                } finally {
                        if (conn != null) {
                                try {
                                        conn.close();
                                } catch (SQLException e) {
                                }
                        }
                }
```

step 2:create configuration file

spring-config.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:jdbc="http://www.springframework.org/schema/jdbc"
        xsi:schemaLocation="http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-4.0.xsd
                http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
        <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/foo" />
                <property name="username" value="root" />
                <property name="password" value="root" />
        </bean>


<bean id="transactionManager"
                class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
                <property name="dataSource" ref="dataSource" />
        </bean>


        <tx:annotation-driven proxy-target-class="true"
                transaction-manager="transactionManager" />
</beans>
```

Now test it

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

        CustomerDao customerDAO = (CustomerDao) context.getBean("customerDAO");
        Customer customer = new Customer("1", "raja",28);
        customerDAO.insert(customer);
```

Using propterties file

db.properties

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/springexp
jdbc.username=root
jdbc.password=root
```

```
<!--    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
                p:location="db.properties"/> -->

                <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
                        <property name="locations" value="classpath:db.properties"></property>
                </bean>

        <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>

        <bean id="customerDAO" class="com.model.CustomerDaoImp">
                <property name="dataSource" ref="dataSource" />
        </bean>
```

Its a pain to write jdbc code!

Ex2:Using jdbcTemplate

Example With JdbcTemplate

With JdbcTemplate, you save a lot of typing on the redundant codes,
becuase JdbcTemplate will handle it automatically.

```
	private DataSource dataSource;
	private JdbcTemplate jdbcTemplate;

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


	public void save(Account account) {

		String sql = "INSERT INTO account (id, name, balance ) VALUES (?, ?, ?)";

		jdbcTemplate = new JdbcTemplate(dataSource);

		jdbcTemplate.update(sql, new Object[]{account.getId(), account.getName(),
account.getBalance()});


	}
```

Ex3:
Example With JdbcDaoSupport

By extended the JdbcDaoSupport, set the datasource
and JdbcTemplate in your class is no longer required,

We just need to inject the correct datasource into
JdbcCustomerDAO. And you can get the JdbcTemplate
by using a getJdbcTemplate() method.

```
	public class AccountDaoImp extends JdbcDaoSupport implements AccountDao {
	  //no need to set datasource here

	  public void save(Account account) {
		String sql = "INSERT INTO account (id, name, balance ) VALUES (?, ?, ?)";
		getJdbcTemplate().update(sql, new Object[]{account.getId(), account.getName(),
account.getBalance()});
		}
	}
```

Ex3:Rowmapper concept

Custom RowMapper

We need to implement the RowMapper interface
to create a custom RowMapper to suit your needs.

Steps:1

Define rowmapper:

```
public class AccountRowMapper implements RowMapper<Account> {

	@Override
	public Account mapRow(ResultSet rs, int arg1) throws SQLException {
		//used to map tuple to oo
		Account account=new Account();
		account.setId(rs.getInt("id"));
		account.setName(rs.getString("name"));
		account.setBalance(rs.getDouble("balance"));
		return account;
	}

}

//http://stackoverflow.com/questions/27591847/how-exactly-work-the-spring-rowmapper-interface
```

step 2:

```
	public Account find(int id) {
		String sql = "SELECT * FROM Account WHERE id = ?";
		Account account = new JdbcTemplate(dataSource).queryForObject(sql,new
AccountRowMapper(), id);
		return account;
	}
```

example: findAll

```
public List<Account> findAll(){
		String sql = "SELECT * FROM Account";
		List<Account> accounts= getJdbcTemplate().query(sql,new
BeanPropertyRowMapper(Account.class));
		return accounts;
}


public List<Account> findAll(){
		String sql = "SELECT * FROM Account";
		List<Account> accounts= getJdbcTemplate().query(sql,new AccountRowMapper());
		return accounts;
}
```

example:update

```
public void update(Account account) {
		String sql = "update account set balance = ? where id=?";
		jdbcTemplate = new JdbcTemplate(dataSource);

		jdbcTemplate.update(sql,new Object[] { account.getBalance(),account.getId() });

	}
```

Java configuration:

```
@Configuration
```

```
@ComponentScan(basePackages={"com.jdbc.bankapp"})
@PropertySource("classpath:db.properties")
public class AppConfig {

	@Autowired
	private Environment env;
	@Bean
	public DriverManagerDataSource getDriverManagerDataSource(){

		DriverManagerDataSource ds=new DriverManagerDataSource();
		ds.setDriverClassName(env.getProperty("jdbc.driverClassName"));
		ds.setUrl(env.getProperty("jdbc.url"));
		ds.setUsername(env.getProperty("jdbc.username"));
		ds.setPassword(env.getProperty("jdbc.password"));
		return ds;
	}
}
```

db.properties

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/foo
jdbc.username=root
jdbc.password=root
```

ref:

---

http://www.mkyong.com/spring/maven-spring-jdbc-example/#
http://www.mkyong.com/spring/spring-jdbctemplate-jdbcdaosupport-examples/

Declerative tx with jdbc

Need?

```
public interface AccountService {
	public void transfer(int fromAccount,int toAccount,double amount);
}


public class AccountServiceImp implements AccountService {

	@Override
	public void transfer(int fromAccount, int toAccount, double amount) {
		Connection connection = null;
		Statement statement=null;
		try{
			Class.forName("com.mysql.jdbc.Driver");
		}catch(ClassNotFoundException ex){}
		try {
```

```
                        connection =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/foo","root","root");
                        connection.setAutoCommit(false);
                  statement = connection.createStatement();
                        statement.executeUpdate("update account set balance = balance - " + amount + "
where id = " + fromAccount);
                        statement.executeUpdate("update account set balance = balance + " + amount + "
where id = " + toAccount);
                        connection.commit();
                } catch (SQLException e) {
                        try {
                                connection.rollback();
                        } catch (SQLException ex) {}
                        throw new RuntimeException(e);
                } finally {
                        try {
                                connection.close();
                        } catch (SQLException ex) {}
                }
        }
}



        AccountService accountService = new AccountServiceImp();
      accountService.transfer(22, 282, 100);
```

Now applying spring based declerative tx

```
@Configuration
@EnableTransactionManagement
//Activating annotation based tx mgt, when @Transactional annotation found, it create a proxy that wrap
actual bean
//when we call business method proxy intercept the call....

@PropertySource("classpath:db.properties")
public class AppConfig {

        @Autowired
        private Environment env;
        @Bean
        public DriverManagerDataSource getDriverManagerDataSource(){

                DriverManagerDataSource ds=new DriverManagerDataSource();
                ds.setDriverClassName(env.getProperty("jdbc.driverClassName"));
                ds.setUrl(env.getProperty("jdbc.url"));
                ds.setUsername(env.getProperty("jdbc.username"));
                ds.setPassword(env.getProperty("jdbc.password"));
                return ds;
        }
```

```
	@Bean
	public PlatformTransactionManager transactionManager() {
		DataSourceTransactionManager transactionManager = new
DataSourceTransactionManager();
		transactionManager.setDataSource(getDriverManagerDataSource());
		return transactionManager;
	}

   @Bean
   public AccountService accountService() {
      AccountServiceImp as = new AccountServiceImp();
      as.setDatasource(getDriverManagerDataSource());
      return as;
   }
}



public class AccountServiceImp implements AccountService {

	private DataSource datasource;

	public void setDatasource(DataSource datasource) {
		this.datasource = datasource;
	}

	@Transactional
	@Override
	public void transfer(int fromAccount, int toAccount, double amount) {
		Connection connection = DataSourceUtils.getConnection(datasource);
		try {
			Statement statement = connection.createStatement();
			statement.executeUpdate("update account set balance = balance - " + amount + "
where id = " + fromAccount);
			statement.executeUpdate("update account set balance = balance + " + amount + "
where id = " + toAccount);
		} catch (SQLException e) {
			throw new RuntimeException(e);
		} finally {
			DataSourceUtils.releaseConnection(connection, datasource);
		}
	}
}


jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/foo
jdbc.username=root
jdbc.password=root
```