Data Access using Spring

spring framework supports for designing and implementing DAO Using JDBC , ORM and Transaction Management

Data Access Objects

“A data access object (DAO) is an object that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, the DAO provides some specific data operations without exposing details of the database.”

Source : Wikipedia

Key points of DAO Pattern

- The Data Access Object (DAO) pattern is a structural pattern that allows us to isolate the application/business layer from the persistence layer (usually a relational database, but it could be any other persistence mechanism) using an abstract API.

- The functionality of this API is to hide from the application all the complexities involved in performing CRUD operations in the underlying storage mechanism. This permits both layers to evolve separately without knowing anything about each other

- The Dao must hide all aspect of communicating with datastore and also aspect related to the data access technologies including exceptions specific to the data access technologies.

Spring Framework DAO Support

- **Consistent exception hierarchy**
  - Spring provides a convenient translation from technology-specific exceptions like SQLException to its own exception class hierarchy with the DataAccessException as the root exception.
- **Annotation used for Configuring DAO or Repository classes**
  - The best way to guarantee that your Data Access Objects (DAOs) or repositories provide exception translation is to use the @Repository annotation.

```
@Repository
public class SomeMovieFinder implements MovieFinder {
        // ...
}
```

More code Snippet…

Any DAO or repository implementation will need to access to a persistence resource, depending on the persistence technology used; for example, a JDBC-based repository will need access to a JDBC DataSource; a JPA-based repository will need access to an EntityManager. The easiest way to accomplish this is to have this resource dependency injected using one of the @Autowired,, @Inject, @Resource or @PersistenceContext annotations Object Relationaal Mapping Support

```
@Repository
public class JdbcMovieFinder implements MovieFinder {
            @Autowired
            private JdbcTemplate   jdbcTemplate;
}
@Repository
public class JpaMovieFinder implements MovieFinder {
            @PersistenceContext
            private EntityManager    entityManager;
}

@Repository
public class HibernateMovieFinder implements MovieFinder {
            @Autowired
             private SessionFactory sessionFactory;
}
```

Spring JDBC Prerequisites

Maven Dependencies
- spring-jdbc
- mysql-connector-java

- Java 8+
- Maven 3+
- Sts 3.5 +
- MySQL
- H2 In Memory DB
- Oracle DB

Spring JDBC Abstraction Framework

The Spring-JDBC component is a part of the Spring framework and is an abstraction on top of the standard Java JDBC API. It takes care of all the low-level API-calls and provides some base classes to implement the DAO-pattern.

Vanilla JDBC vs Spring JDBC

Action	Spring	You
Define connection parameters		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

Choosing an Approach for JDBC Database Access

SqlUpdate

MappingSqlQuery

NamedParameterJdbcTemplate

StoredProcedure

SimpleJdbcInsert

SimpleJdbcCall

JdbcTemplate

JdbcTemplate

The central class of the spring Jdbc abstraction framework.

When you use the JdbcTemplate for your code, you only need to implement callback interfaces, giving them a clearly defined contract.

JdbcTemplate is Threadsafe

Controlling database connections and DataSource

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

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.

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, Then provide a username and a password to connect to the database

DataSource cntd…

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

DriverManagerDataSource Configuration:

```
<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>
```

DataSource cntd…

DBCP configuration:

```
<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>
```

C3Po configuration:

```
<bean id="dataSource" class="com.mchange.v2.c3po.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>
```

Embedded database support

The org.springframework.jdbc.datasource.embedded package provides support for embedded Java database engines. Support for HSQL, H2, and Derby is provided natively. You can also use an extensible API to plug in new embedded database types and DataSource implementations.

Creating an embedded database using Spring XML

```
<jdbc:embedded-database id="dataSource" generate-name="true">
        <jdbc:script location="classpath:schema.sql"/>
        <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
```

Creating an embedded database programmatically

```
EmbeddedDatabase db = new EmbeddedDatabaseBuilder()
        .generateUniqueName(true)
        .setType(H2)
        .setScriptEncoding("UTF-8")
        .ignoreFailedDrops(true)
        .addScript("schema.sql")
        .addScripts("user_data.sql", "country_data.sql")
        .build();
```

JNDI data source

- **Add following in tomcat context.xml**

```
<Resource name="jdbc/spring" auth="Container"
          type="javax.sql.DataSource"
          maxTotal="100" maxIdle="30" maxWaitMillis="10000"
          username="system" password="pratap"
          driverClassName="oracle.jdbc.driver.OracleDriver"
           url="jdbc:oracle:thin:@localhost:1521:orcl"/>
```

  – https://tomcat.apache.org/tomcat-9.0-doc/jndi-datasource-examples-howto.html

JNDI data source

- **Edit the following in your application web.xml**

```
<resource-ref>
        <description>DB Connection</description>
        <res-ref-name>jdbc/spring</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
</resource-ref>
```

- **Use The following in applicationcontext.xml**

```
<bean id="template" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="ds"/>
</bean>
<jee:jndi-lookup jndi-name="jdbc/spring" id="ds”/>
```

Jndi

Updating the database

- public int update(java.lang.String sql) throws DataAccessException

- public int update(java.lang.String sql, java.lang.Object... args) throws DataAccessException

- public int update(java.lang.String sql, java.lang.Object[] args, int[] argTypes)throws DataAccessException

- public int update(java.lang.String sql, PreparedStatementSetter pss) throws DataAccessException

- public int update(PreparedStatementCreator psc)

  throws DataAccessException

- public int update(PreparedStatementCreator psc, KeyHolder generatedKeyHolder)throws DataAccessException

Running queries

Executing statements

Batch operations with a List of objects

```
public int[] batchUpdate(final List<Actor> actors) {
SqlParameterSource[] batch = SqlParameterSourceUtils.createBatch(actors.toArray());
int[] updateCounts = namedParameterJdbcTemplate.batchUpdate(
"update t_actor set first_name = :firstName, last_name = :lastName where id = :id",
batch);
return updateCounts;
}
```

Retrieving auto-generated keys

- An update() convenience method supports the retrieval of primary keys generated by the database.
- This support is part of the JDBC 3.0 standard;
- The method takes a PreparedStatementCreator as its first argument, and this is the way the required insert statement is specified.
- The other argument is a KeyHolder, which contains the generated key on successful return from the update. There is not a standard single way to create an appropriate PreparedStatement

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.

An instance of this class is thread-safe once configured.

SQLException Translator

- SQLExceptionTranslator is an interface to be implemented by classes that can translate between SQLExceptions and Spring’s own org.springframework.dao.DataAccessException

- SQLErrorCodeSQLExceptionTranslator is the implementation of SQLExceptionTranslator that is used by default. This implementation uses specific vendor codes.

- The error code translations are based on codes held in a JavaBean type class called SQLErrorCodes. This class is created and populated by an SQLErrorCodesFactory which as the name suggests is a factory for creating SQLErrorCodes based on the contents of a configuration file named sql-error-codes.xml.

Simplify Jdbc Operation

The SimpleJdbcInsert and SimpleJdbcCall classes provide a simplified configuration by taking

advantage of database metadata that can be retrieved through the JDBC driver.

SimpleJdbcInsert

SimpleJdbcCall

SqlParameterSrouce

Spring ORM Integration Support

The Spring Framework supports integration with the Java Persistence API (JPA) and supports native Hibernate for resource management, data access object (DAO) implementations, and transaction strategies.

ORM Integration

- For Hibernate, there is first-class support with several convenient IoC features that address many typical Hibernate integration issues.
- You can configure all of the supported features for OR (object relational) mapping tools through Dependency Injection.
- They can participate in Spring’s resource and transaction management, and they comply with Spring’s generic transaction and DAO exception hierarchies.
- The recommended integration style is to code DAOs against plain Hibernate or JPA APIs.

General ORM Integration Consideration

- Resource and Transaction Management
  - Spring advocates simple solutions for proper resource handling, namely IoC through templating in the case of JDBC and applying AOP interceptors for the ORM technologies.
- Exception Translation
  - When you use Hibernate or JPA in a DAO, The DAO throws a subclass of a HibernateException or PersistenceException, depending on the technology.
  - Spring lets exception translation be applied transparently through the @Repository annotation.

```
@Repository

public class ProductDaoImpl implements ProductDao {   }

<bean
class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>
```

Integration with Hibernate

- SessionFactory Setup in a Spring Container
  - To avoid tying application objects to hard-coded resource lookups, you can define resources (such as a JDBC DataSource or a Hibernate SessionFactory) as beans in the Spring container.
  - use LocalSessionFactoryBean and LocalSessionFactoryBuilder for xml and java based configuration respectively.
- Implementing DAOs Based on the Plain Hibernate API
- Spring provides a Template variant for Hibernate named HibernateTemplate like JdbcTemplate . However the usage of HibernateTemplate is discouraged .

Transaction Management

Comprehensive transaction support is among the most compelling reasons to use the Spring Framework.

Transaction Primer

- A transaction symbolizes a unit of work performed within a database management system (or similar system) against a database, and treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in a database.
- A database transaction, by definition, must be atomic, consistent, isolated and durable.

- Global Transaction is an application server managed transaction, allowing to work with different transactional resources (this might be two different database, database and message queue, etc)
- Local Transaction is resource specific transaction (for example Oracle Transactions) and application server has nothing to do with them

Spring Transaction Benefit

- A consistent programming model across different transaction APIs, such as Java Transaction API (JTA), JDBC, Hibernate, and the Java Persistence API (JPA).
- Support for declarative transaction management.
- A simpler API for programmatic transaction management than complex transaction APIs, such as JTA.
- Excellent integration with Spring's data access abstractions.

Spring Framework’s Transaction Support

- Java EE developers have had two choices for transaction management: global or local transactions, both of which have profound limitations.

- Local transactions are resource-specific, such as a transaction associated with a JDBC connection. Local transactions may be easier to use but have a significant disadvantage: They cannot work across multiple transactional resources.

- Global transactions let you work with multiple transactional resource , The application server manages global transactions through the JTA, which is a cumbersome API , a JTA UserTransaction normally needs to be sourced from JNDI, meaning that you also need to use JNDI in order to use JTA.

Consistent Tx API

- Spring resolves the disadvantages of global and local transactions. It lets application developers use a consistent programming model in any environment.

- The Spring Framework provides both declarative and programmatic transaction management.

- Do you need an application server for transaction management?

Spring Framework Transaction Abstraction

- **PlatformTransactionManager**

Programmatic Tx Management

- PlatformTransactionManager

- TransactionTemplate

Declarative Tx Management

- TransactionProxyFactoryBean (legacy)
- XML based configuration
  - Using tx namespace
- Annotation based configuration
  - @Transactional

TransactionProxyFactoryBean

```
<bean id="baseTransactionProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
    abstract="true">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributes">
    <props>
      <prop key="insert*">PROPAGATION_REQUIRED</prop>
      <prop key="update*">PROPAGATION_REQUIRED</prop>
      <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
 </bean>
```

Using <tx> namespace

- <tx:advice/>
  - Transactional settings that can be specified using the <tx:advice/> tag.

```
<tx:advice id=""  transaction-manager="transactionManager">
        <tx:attributes > <!--  No attributes -- >
        <tx:method name="" isolation="DEFAULT"
                                        no-rollback-for=""
                                        propagation="REQUIRED"
                                         read-only="false"
                                        rollback-for=""
                                        timeout="-1"/>
        <tx:method name=""/>
        </tx:attributes>
</tx:advice>
```

Using <tx> namespace

- ensure that the above transactional advice runs for any execution of an operation defined by the FooService interface

```
<aop:config>

        <aop:pointcut id="fooServiceOperation"
                expression="execution(* x.y.service.FooService.*(..))"/>

        <aop:advisor advice-ref="txAdvice"
                        pointcut- ref="fooServiceOperation"/>

</aop:config>
```

- In the above configuration , You want to make a service object, the fooService bean, transactional. The transaction semantics to apply are encapsulated in the <tx:advice/> definition.

Using @Transactional

- In addition to the XML-based declarative approach to transaction configuration, you can use an annotation-based approach.
- Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code.
- There is not much danger of undue coupling, because code that is meant to be used transactionally is almost always deployed that way anyway.

- <!-- enable the configuration of transactional behavior based on annotations -->
  - <tx:annotation-driven transaction-manager="txManager"/>
  - @EnableTransactionManagement annotation provides equivalent support if you are using Java based configuration.

Thank You!

Q & A