Java Data Access Object (DAO)

“Just the facts

Relational Data Base Access from Java

Two common techniques:

- **Native JDBC** - Programmer codes performs all function and writes all statements to interact and manipulate the database.

- **Spring JDBC Framework** - Programmer defines classes (Model), interfaces and concrete classes for the database then instantiates and uses the objects to manipulate the database.
  - The framework takes care of the details of interacting with the database
  - Programmer concentrates on manipulating the database for the application
  - Programmer must learn the framework and how to interact with it

Java Spring DAO - Just the facts

Data **A**ccess **O**bject - separate low level data accessing API operations from high level business services

Major components:

- **Model Object** (Value Object) - This object is simple POJO containing **get/set methods to retrieve/store data** using DAO class. Create one class for each table to be accessed.
- **DAO Interface** - This interface **defines the standard operations** (CRUD) be performed on a model object(s). Create one interface for each table being accessed.
- **DAO concrete class** - This class **implements DAO interface**. This class is responsible for all data access from a data source (database, text file, xml, json, et al). Create one concrete class for each DAO interface.
- Place all components for a table in the same Java package.

API = **A**pplication **P**rogramming **I**nterface
POJO = **P**lain **O**ld **J**ava **O**bject
CRUD = **C**reate, **R**ead, **U**pdate, **D**elete
JDBC - **J**ava **D**ata **B**ase **C**onnectivity

Spring JDBC Framework - Just the facts

Spring JDBC Framework - takes care of the low-level details that can make JDBC "tedious".

Major components:

- **JDBCTemplate** - Central class in the JDBC core package. It simplifies the use of JDBC since it handles the creation and release of resources, SQL queries, update statements, catches JDBC exceptions and translates them to the generic, more informative messaging.

```
JdbcTemplate myJdbcTemplate = new JdbcTemplate(aDataSource) ;
```

- **SqlRowSet** - Data structure to hold the result of an SQL query. A cursor (pointer) the current row is maintained. `SqlRowSet results ;`

- When creating the SQL statement, place **?** in place of literals where values are desired in the SQL statement. Values for the placeholders are specified when the statement is executed

- **queryForRowSet()** - Execute a query returning the result in an **SqlRowSet.**

```
String sqlGetCityInfo = "SELECT id, name, countrycode FROMcity where id = ?" ;
results = myJdbcTemplate . queryForRowSet(sqlGetCityInfo, 319 ) ;
```

- **update()** - Execute a SQL INSERT, UPDATE or DELETE statement.

```
String sqlAddCity = "INSERT INTO city (name,countrycode, district,population) "
                    +"VALUES(?,?,?,?") ;
myJdbcTemplate . update(sqlAddCity,newName, country, state, pop) ;
```

Some SqlRowSet methods

ResultSet retrieval methods *(one for each datatype, only a few shown)*:

next() - move the cursor to next row; return true if next row exists, false otherwise

getInt("*column - name*") - retrieve value for *column-name* in current row as an int
getInt("*column - number*") - retrieve value for *column-number* in current row as an int

getString("*column - name*") - retrieve value for *column-name* in current row as a String object
getString("*column - number*") - retrieve value for *column-number* in current row as a String object

getDate("*column - name*") - retrieve value for *column-name* give in current row as a Date using object
getDate("*column - number*") - retrieve value for *column-number* given in current row as a Date object
getDate("*column - name*").toLocalDate() - retrieve value for *column-name* in current row as a LocalDate
getDate("*column - name*").toLocalDate() - retrieve value for *column-number* in current row as a LocalDate

column-number specification is a **1-based** value (first column is designated as 1)

Specifying incorrect datatype for the column value will result in an SQLException

Link to information on using Java Date objects: https://www.tutorialspoint.com/java/java_date_time.htm

JDBC Application Program Components - Just the facts

DataSource - Class to define the data base as an object. Used to connect Spring JDBC/DAO to the data base. Several data source classes are compatible for Spring JDBC/DAO

1. Define a BasicDataSource object:

```
BasicDataSource aDataSource = new BasicDataSource();
aDataSource.setUrl("jdbc:database-mgr://localhost:port/database-name");
aDataSource.setUsername("db-owner-id");
aDataSource.setPassword("db-owner-pswd");
```

1. Define an object for the DAO you wish to use passing it the datasource for the data base.

```
ActorDao anActorDao = new actorDAO(aDataSource) ;
```

1. Use the methods of the DAO passing any parameters required by the method. Store any value returned from the method.

```
List<Film> actorFilms = anActorDAO.getFilmsForActor("Nick", "Stallone");
```

A word concerning using PostgreSQL date/time types in Java

Java 8 provides a new set of classes to easily use PostgreSQL dates in Java:

PostgreSQL™	Java SE 8
DATE	LocalDate
TIME [WITHOUT TIMEZONE]	LocalTime
TIMESTAMP [WITHOUT TIMEZONE]	LocalDateTime
TIMESTAMP WITH TIMEZONE	OffsetDateTime

LocalDate - an immutable date/time object in the format **yyyy-mm-dd**

LocalTime - an immutable date/time object in the format **hh:mm:ss**

LocalDateTime - an immutable date/time object in the format **yyyy-mm-ddThh:mm:ss**

OffsetDateTime - an immutable date/time object with an offset from UTC/GMT:
yyyy-mm-ddThh:mm:ss+h:mm

For additional information and methods, visit the Oracle Java Technical information sites:

https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html
https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html
https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html
https://docs.oracle.com/javase/8/docs/api/java/time/OffsetDateTime.html

Using Spring JDBC DAO - What you need

For <u>each</u> table you want to access from an application program you need:

1. A **Model Object** - a POJO to describe the data in the table:
    a. File name should be *table-name*.java
    b. Class name name should be the same as table name
    c. Good idea if instance variable names match column names, but not required
    d. Be sure you have correct data types for the columns
    e. Standard getters and setters

1. **A DAO interface** identifying the methods required for table access:
    a. File and Class name should be *table-name*DAO.java
    b. Should define at least CRUD methods

1. A **DAO Concrete Class** implementing the methods required by the DAO interface:
    a. File and Class name should be: Jdbc*Table-name*DAO.java
    b. Define a **JdbcTemplate** object reference to interact with Spring DAO methods.
    c. Code a constructor for the concrete class that takes a data source as a parameter.
    d. Instantiate a **JdbcTemplate** object using the data source passed to the constructor and assign it to the **JdbcTemplate** reference defined in step a.

Some JDBC ResultSet methods

ResultSet Navigation methods:

next() - move to next row in ResultSet; returns false if no next row

previous() - move to previous row in ResultSet; returns false if no previous row;

Requires ResultSet.TYPE_SCROLL_INSENSITIVE or ResultSet.TYPE_SCROLL_SENSITIVE specified in call to createStatement() method:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE);
```

ResultSet.TYPE_SCROLL_INSENSITIVE - Result is not sensitive to changes made to the database by others after the result was created

ResultSet.TYPE_SCROLL_SENSITIVE - Result IS sensitive to changes made to the database by others after the result was created

first() - move to first row in ResultSet row

last() - move to last row in ResultSet row

Some JDBC ResultSet methods

ResultSet retrieval methods *(one for each datatype, only a few shown)*:

`getInt("column-name")` - retrieve value for *column-name* give in current row as an int

`getInt("column-number")` - retrieve value for *column-number* given in current row as an int

`getString("column-name")` - retrieve value for *column-name* give in current row as a String object

`getString("column-number")` - retrieve value for *column-number* given in current row as a String object

`getDate("column-name")` - retrieve value for *column-name* give in current row as a Date using object

`getDate("column-number")` - retrieve value for *column-number* given in current row as a Date object

column-number specification is a **1-based** value (first column is designated as 1)

Specifying incorrect datatype for the column value will result in an SQLException

Link to information on using Java Date objects: https://www.tutorialspoint.com/java/java_date_time.htm

JDBC Prepared Statements and Parameters

SQL statements may be prepared for execution with parameter markers for values rather than literals for values:

1. Define a PreparedStatement object for the statement;
2. Use the connection object to run the prepareStatement() method for the SQL statement.
3. Place **?** in place of literals where values are desired in the SQL statement.

```
PreparedStatement findActorByNameStmt =
        conn.prepareStatement("SELECT * FROM actor WHERE first_name = ? AND last_name = ?");
```

1. Use the SQL statement to run the setDatatype() method to assign actual values to the parameter markers (?) in the SQL statement. The setDatatype() method takes the parameter number and value to be assigned the parameter marker:

```
findActorByNameStmt.setString(1, actorFirstName);
findActorByNameStmt.setString(2, actorLastName);
```

Note: Be sure you have specified the column number and value pair that matches the parameter marker you want to replace.

1. Use the SQL statement to run the executeQuery() method

```
results = findActorByNameStmt.executeQuery();
```