Database Connectivity Architecture for Automation

Table Object Approach

The database connectivity Architecture for connected Automation, from out utility package we will have folder called “jdbc” where the Java Database Connectivity classes resides.

**Util Classes:**

1. **DatabaseServer :**

Since we have mirrored/clustered database architecture it’s possible to have different database servers reside in different machines, also we have a light weight DB (sqllite DB) for connected Agent, so same test case may need to connect to different databases. In such cases we may need to specify the connection type from the caller. So it’s easy for the connection manager to establish the connection based on the connection type. DatabaseServer is of type enum, where the user should specify which type of connection the user is going to make. In other words this specifies which database server the user is going to make a connection.

Examples

COMMON\_SERVER - if the test cases requires only one standalone database server.

PRIMARY\_SERVER - if the test case needs to verify information on primary database server.

SQLLITE\_DB - if the test case is an agent test and need to verify some information from Sqllite DB.

If the test case needs to establish a connection to a new database server, then the user can create a new enum of type “DatabaseServer”.

The DatabaseServer will load the appropriate connection properties from the databaseconfig file (jdbcconfig.xml)

1. **ConnectionManager :**

Since we have to connect to different database servers it’s always a good idea to segregate the code to establish the database connection for maintainability. The ConnectionManager will take the DatabaseServer and database name as arguments and will return the connection object.

Here we will establish a database connection and returns the connection object.

Connections are created using apache commons.pool to implement connection pooling.

ConnectionManager will have methods to set all the properties directly. Also a number of constructors which sets the properties using DatabaseServer or directly.

.

1. **QueryExecutor:**

QueryExecutor class which takes the connection object and executes the queries on it(executeQuery/executeUpdate) and returns result set.

(Query Manager should implement in such a way that it can handle time out issues)

QueryExecutor has many constructors where it can directly get connected to a specific database server using DatabaseServer type, or giving the ConnectionManager object to establish a connection using the connectionManager.

1. **ResultSetManager:**

ResultSetManager class which does the meta/common operations over the result set.

This class is basically intended to work with the test cases (Verifications). Probably need to do the operations like getStringColumnValues(ResultSet resultSet, String columnLabel) which will return the values of the particular column. So the caller (test Case) can verify that the value of a particular column is correct.

**Library classes:**

Will create an object representation of each tables in the databases, for connected will have 2 databases so the structure will be

com.imd.connected.library.database.registry->CustomerTable.java, CdqTable.java, CommunityTable.java etc..

com.imd.connected.library.database. directory ->FileIndexTable.java, PoolIndexTable.java etc..

The table object will have the methods to get the table data information, and the queries will be placed in this level, so if any change in the table level we can easily go the table object and can modify accordingly. (This is the java DAO layer, we term as TAOs Table Access Objects, since we are mapping the action on Tables not the actual objects).

In practice the test cases say go to the CDQ table and verify the entry, in this cases use the CDQTable object’s getters and setters to verify the data.

For example

CdqTable.java will have methods like

getNumberOfDvdOrders(); will get from ResultSetManager.getNumberOfRows(String columName, dataValue)

Sample Hierarchy For Connected Application:

* All the databases must extend from Database class (Database has many methods and properties which can used across the tables and databases)

**Sample Table Object Code:**

public class CdqTable extends RegistyDatabase{

/\*

\*Note that the queryExecutor and DATABASE\_NAME are declared in Database, when we create a Table Object, will construct a QueryExecutor for the table.

Here I am using a DatabaseServer type to get the connection properties

public CdqTable(DatabaseServer dbServer){

setDatabaseServer(dbServer);

queryExecutor = new QueryExecutor(databaseServer, DATABASE\_NAME);

}

/\*it’s Possible to construct a Table Object using the ConnectionManager, Here the connection will be established using the custom ConnectionManager and the QueryExecutor for the table will be returned.

public CdqTable(ConnectionManager conMgr){

queryExecutor = new QueryExecutor(conMgr, DATABASE\_NAME);

}

/\* Executing the queries over the queryExecutor.

public int[] getAccountNumbers(){

queryExecutor.executeAccountQuery(); }

}

Sample Test to make use of the Table Object and JDBC Classes refractor

Public class TestingJDBC{

CdqTable cdqTable = null;

@test

Public void testCdqTableWithCommonDatabase(){

cdqTable = **new** CdqTable(DatabaseServer.*COMMON\_SERVER*);

Asserter.*assertEquals*(cdqTable.getAccountNumber()), accountNo); Asserter.*assertEquals*(cdqTable.getMediaType(), mediaType); Asserter.*assertEquals*(cdqTable.getShippingLabel(), shippingLabel); cdqTable.closeQueryExecutor(); cdqTable.closeDatabaseConnection();

}

@test

Public void testCdqTableWithCustomDatabase(){

/\* creating connection using specific properties…

Note that the constructor used will take the common properties like servertype, driver, and port.

We have constructors where we can pass all the connection properties.

ConnectionManager conMgr = new ConnectionManager(serverUrl, RegistryDatabase.NAME, username, password,);

cdqTable = **new** CdqTable(ConMgr);

Asserter.*assertEquals*(cdqTable.getAccountNumber()), accountNo); Asserter.*assertEquals*(cdqTable.getMediaType(), mediaType); Asserter.*assertEquals*(cdqTable.getShippingLabel(), shippingLabel); cdqTable.closeQueryExecutor(); cdqTable.closeDatabaseConnection();

}

}

**Config File:**

Since we will have more than 1 database server, we need to provide the available database server information to the framework using the “jdbcconfig.xml”. Basically the user should define all the properties required for all the DatabaseServer’s. A sample Config file looks like this,

// Configuration Properties for Connection type COMMON\_SERVER. If the application is not a mirrored or clustered one and it has only one database server then the user can define this properties and use the DatabaseServer as COMMON\_SERVER.

<!-- Properties required to connect to the database -->

<DBDriver>com.microsoft.sqlserver.jdbc.SQLServerDriver</DBDriver>

<!-- DBServerType represents the Database Server in use, Values should be taken from(MSSQL, Oracle, SQLLite, MySQL...) -->

<DBServerType>MSSQL</DBServerType>

<DBServerUrl>jdbc:sqlserver://conqasvrin01.calpurnia.com</DBServerUrl>

<DBPort>1433</DBPort>

<DBUsername>sa</DBUsername>

<DBPassword>sa</DBPassword>

<PrimaryServerDBDriver>com.microsoft.sqlserver.jdbc.SQLServerDriver</PrimaryServerDBDriver>

<PrimaryServerType>MSSQL</PrimaryServerType>

<PrimaryServerDBServerUrl>jdbc:sqlserver://vconqadcin.calpurnia.com</PrimaryServerDBServerUrl>

<PrimaryServerDBPort>1433</PrimaryServerDBPort>

<PrimaryServerDBUsername>sa</PrimaryServerDBUsername>

<PrimaryServerDBPassword>sa</PrimaryServerDBPassword>

<SecondaryServerDBDriver>com.microsoft.sqlserver.jdbc.SQLServerDriver</SecondaryServerDBDriver>

<SecondaryServerType>MSSQL</SecondaryServerType>

<SecondaryServerDBServerUrl>jdbc:sqlserver://conqamdcin.calpurnia.com</SecondaryServerDBServerUrl>

<SecondaryServerDBPort>1433</SecondaryServerDBPort>

<SecondaryServerDBUsername>sa</SecondaryServerDBUsername>

<SecondaryServerDBPassword>sa</SecondaryServerDBPassword>