IBM Db2 Java Reactive Driver

Developer Guide

Version 1.1.0

1-24-2023

Contents

1. Introduction

This document describes how to use the Java Reactive Database Driver for Db2, various usage scenarios and source code samples. Refer to the "Installation Guide" to install and setup the Reactive Driver environment before trying the examples described in this document.

Following below is the outline of the topics discussed in this document:

1) Building Connection Details
2) Creating Connection Factory
3) Getting Database Connection
4) Creating Connection Pool
5) Using a Pooled Connection
6) Executing SQL Statements
7) Processing Result Set
8) Statement Caching
9) Transactions
10) Data Type Mapping
11) Supported Data Types

2. Connection Details

DB2ConnectionConfiguration object is used to hold the connection configuration details. This is a read only object built using the Builder subclass as show in the example below. The factory method builder() is used to create an instance of Builder class whose methods are then used to set the connection configuration parameters. At the end, build() method is used to create instance of DB2ConnectionConfiguration object.

```
DB2ConnectionConfiguration config = DB2ConnectionConfiguration.builder()
                    .database("mydb")
                    .host("myhost")
                    .port(1234)
                    .username("ts1234")
                    .password("mypassword")
                    .conPoolSize(100)
                    .stmtCacheSize(30)
                    .build();
```

Below is the list of connection configuration parameters supported.

Property Name	**Property Description**
database	Name of the database on the server to connect to.
host	Name of the server to connect to.
port	Port number on which the database server listens on.

enableSSL	Enables SSL support when the parameter is true. Default value is false. Currently SSL is supported for Server Authentication mode only. No Client Authentication and Mutual Authentication support.
trustStorePath	Used to provide the path to SSL Trust Store file. The trust store is expected to be in JKS file format with Server Certificate imported in it.
trustStorePassword	Used to provide the password for SSL Trust Store.
securityMechanism	Security mechanism used to authenticate and authorize a user. Supported security mechanisms are: 1. `USERID_PASSWORD` `2.KERBEROS_SECURITY`
kerberosServerPrincipal	When using Kerberos Security mechanism, use this property to specify the Kerberos Server Principal of the database server.
username	User-id to be used for authentication.
password	User password.
conPoolSize	Max number of connections in the connection pool. Default value is 10.
stmtCacheSize	Max number of statements to be cached in a connection. Default value is 30.
isolation	Sets the transaction isolation level for the connections. Default is Read-Committed.
fetchSize	Number of rows to fetch from the server in one go. Default is 100.
keepDynamic	A Boolean value ‘true’ or ‘false’, used during the binding of packages used on server. Default is false.

3. Connection Factory

After building the DB2ConnectionConfiguration object, one can instantiate the DB2ConnectionFactory object. This object is a factory object to create connections to the Db2 database. Below is a code example.

```
DB2ConnectionFactory factory = new DB2ConnectionFactory(config);
```

4. Getting Database Connection

When we have the DB2ConnectionFactory object, we can create connections to the database using it’s create() method. The create() method returns a Mono, a publisher that can publish one DB2Connection object when somebody has subscribed to this publisher. Below is a code example.

```
Mono<DB2Connection> mono = factory.create();
```

5. Creating Connection Pool

When we have the DB2ConnectionFactory instance, this can create DB2Connection objects. Using this factory object, we can create a connection pool. We can instantiate DB2ConnectionPool using DB2ConnectionFactory instance. When connection pool is instantiated, it starts to populate its connection pool with DB2Connection objects until the user specified conPoolSize in DB2ConnectionConfiguration is reached. User can then use getConnection() method is connection pool object to get a free connection. Below is the code example to instantiate a DB2ConnectionPool.

```
DB2ConnectionPool pool = new DB2ConnectionPool(factory);
```

6. Using Pooled Connection

To get a free connection from DB2ConnectionPool, user must invoke the `getConnection()` method. This method returns a Mono that when subscribed to, will publish a DB2Connection object. When user is done using the connection object and wants to release the connection back to the pool, user must invoke the `release()` method on the DB2Connection object. Below is a code sample.

```
AtomicReference<DB2Connection> con = new AtomicReference<DB2Connection>();

Mono<Void> mono = pool.getConnection()
            .doOnNext(c -> con.set(c))
            .flatMap(c -> c.executeUpdate(
                        "CREATE TABLE DEMO (" +
                              "ID INTEGER UNIQUE NOT NULL, " +
                              "ACC_BAL DECIMAL(10,2) NOT NULL, " +
                              "AS_OF TIMESTAMP NOT NULL" +
                        ")"
            ))
            .doOnTerminate(() -> con.get().release());
```

When done using the connection pool or when the application is shutting down, user must shutdown the DB2ConnectionPool. Invoking closeAll() method on the connection pool object results in closing of all the physical connections to the database and shutting down the connection pool.

```
// close all the connections in the pool
Mono<Void> mono = pool.closeAll();
```

The Mono returned indicates the completion status of the operation.

7. Executing SQL Statements

After getting a connection successfully you can execute SQL statements and process the results. Each of the subsections below shows execution of different type of SQLs, preparing, executing and processing them.

a) Execute Immediate SQL

Simple one-time execution of SQLs typically of DDL type can be executed using execute-immediate feature. The result of the execution will indicate a success or failure of the SQL execution and the number of rows updated in the case of an update statement. Below is a code sample.

```
AtomicReference<DB2Connection> con = new AtomicReference<DB2Connection>();

Mono<Void> mono = pool.getConnection()
       .doOnNext(c -> con.set(c))
       .flatMap(c -> c.executeUpdate("DELETE FROM PHASE1_DEMO VALUES"))
       .doAfterTerminate(() -> con.get().release());
```

b) Prepare SQL

Frequently used SQLs can be prepared and cached once and the same can be executed many times using different parameter values. To do this, first user must turn on Statement Caching feature by using a positive integer value for the stmtCacheSize in DB2ConnectionConfiguration object. This value indicates the number of statements that will be cached in this connection. When a prepared statement is executed for the first time and when the SQL is prepared, the prepared statement object will be cached in memory. Below is a code example.

```
AtomicReference<DB2Connection> con = new AtomicReference<DB2Connection>();

Mono<DB2Result> mono = pool.getConnection()
       .doOnNext(c -> con.set(c))
       .flatMap(c -> c.createStatement("INSERT INTO PHASE1_DEMO (ID, ACC_BAL,
                           AS_OF) VALUES (?, ?, ?)")
                                  .bind(1, id)
                                  .bind(2, amount)
                                  .bind(3, Timestamp.from(Instant.now()))
                                  .execute())
       .doAfterTerminate(() -> con.get().release());
```

Alternatively, user can also prepare the SQL upfront even before its first execution and cache it using the prepare() method on the DB2PreparedStatement. This can be done during the DB2Connection instantiation time by overriding the init() method of DB2ConnectionFactory. Whenever a connection is created by the factory, it will call the init() method for any one-time initializations to be done on that connection. Below is a code example.

```
// Define the SQLs whose prepared statements you want to cache
public static class SQL {
 // All SELECT, INSERT, UPDATE and DELETE statements can be cached
 // Following statements assume a DEMO table with ID, BAL columns.
 public static final String ADD = "INSERT INTO DEMO (ID, BAL) VALUES (?, ?)";
 public static final String REMOVE = "DELETE FROM BAL WHERE ID = ?";
 public static final String CREDIT = "UPDATE DEMO SET BAL = BAL + ? WHERE ID = ?";
 public static final String DEBIT = "UPDATE DEMO SET BAL = BAL - ? WHERE ID = ?";

 // SET statements will not be cached. They can be invoked once when a new
 // connection is created.
 public static final String SET_SCHEMA = "SET CURRENT SCHEMA TS1234";
```

```
}

// Define a custom connection factory
public static class MyConnectionFactory extends DB2ConnectionFactory {
   // Constructor
   public MyConnectionFactory(DB2ConnectionConfiguration configuration) {
       super(configuration);
   }

   // Override the init method, this method will be called once for each new
   // connection instance
   public Mono<Void> init(DB2Connection con) {
       // perform all the initialization tasks to be done for the connection
       return Mono.just(con)
                  .delayUntil(c -> c.executeUpdate(SQL.SET_SCHEMA))
                  .delayUntil(c -> c.createStatement(SQL.ADD).prepare())
                  .delayUntil(c -> c.createStatement(SQL.REMOVE).prepare())
                  .delayUntil(c -> c.createStatement(SQL.CREDIT).prepare())
                  .delayUntil(c -> c.createStatement(SQL.DEBIT).prepare())
                  .then();
   }
}

// use the new custom factory
MyConnectionFactory factory = new MyConnectionFactory(config);
```

c) Execute Update SQL

SQL statements can be executed by invoking the execute() method on the DB2PreparedStatement object. The returned DB2Result object will return the number of rows updated when its `getNumRowsUpdated()` method is called. Below is code example.

```
    AtomicReference<DB2Connection> con = new AtomicReference<DB2Connection>();

    Mono<DB2Result> mono = _pool.getConnection()
           .doOnNext(c -> con.set(c))
           .flatMap(c -> c.createStatement(SQL.ADD)
                                  .bind(1, id)
                                  .bind(2, amount)
                                  .execute())
           .doOnNext(result -> logger.debug(("Updated "+
                                 result.getNumRowsUpdated()+" row(s).")
           .doAfterTerminate(() -> con.get().release());
```

The createStatement() method looks for the availability of a prepared statement with same SQL text in the prepared statement cache. If one is available, it will be returned by the createStatement() method. Otherwise a new instance will be instantiated, prepared and added to the cache.

d) Execute Query SQL

SQL Query statements can be executed by invoking the same execute() method on the DB2PreparedStatement object. The returned DB2Result can then be used to retrieve the rows and the

column values in each of the rows. Below is a code example that shows how to retrieve column values for each of the rows.

```
AtomicReference<DB2Connection> con = new AtomicReference<DB2Connection>();

// Query the inserted data and then release the connection
Flux<Row> flux = pool.getConnection()
       .doOnNext(c -> con.set(c))
       .flatMap(c -> c.createStatement("SELECT ID, ACC_BAL, AS_OF FROM “+
                                       “PHASE1_DEMO WHERE ID IN (?, ?)")
           .bind(1, src)
           .bind(2, dest)
           .execute())
           .flatMapMany(result -> result.map((row, md) -> row))
           .doOnNext(row -> logger.debug("ID: "+row.get("ID")+", ACC_BAL: "
                        +row.get("ACC_BAL")+", AS_OF: "+row.get("AS_OF")))
           .doOnError(e -> logger.error("Error querying account details - "+e))
           .doAfterTerminate(() -> con.get().release());
```

As usual, remember to release the connection back to connection pool after done.

e) Map a Row to an Object (ORM)

When we receive a Row object in the DB2Result map() method we also have an option to map each of the returned row to an Object representing the row. Code example is shown below.

```
/**
 * Represents a row in the DEMO table.
 * Object Relational Mapping.
 */
public static class Account
{
       private Integer _id;
       private BigDecimal _accBal;
       private Timestamp _asOf;

       public Account(Row row)
       {
              _id = (Integer)row.get("ID");
              _accBal = (BigDecimal)row.get("ACC_BAL");
              _asOf = (Timestamp)row.get("AS_OF");
       }

       public Integer getAccountId() {return _id;}
       public BigDecimal getAccountBalance() {return _accBal;}
       public Timestamp getAsOf() {return _asOf;}

       public String toString()
       {
              return "Account {id: "+_id+", acc_bal: "+_accBal+
                              ", as_of: "+_asOf+"}";
       }
}
```

```
/**
 * Selects all the rows from the table DEMO and maps each of the row
 * to an Account object.
 */
public Flux<Account> queryAllAccounts()
{
    _logger.debug("queryAllAccounts() called");

    AtomicReference<DB2Connection> con = new AtomicReference<DB2Connection>();

    // Query for all rows and then release the connection
    Flux<Account> flux = _pool.getConnection()
           .doOnNext(c -> con.set(c))
           .flatMap(c -> c.createStatement("SELECT ID, ACC_BAL, AS_OF FROM DEMO")
                          .execute())
           .flatMapMany(result -> result.map((row, md) -> new Account(row)))
           .doOnError(e -> _logger.error("Error querying all accounts details - "
                                         +e))
           .doAfterTerminate(() -> con.get().release());

           return flux;
}
```

8. Statement Caching

Statement caching can be enabled by specifying a positive integer value greater than zero for the stmtCacheSize parameter in the DBConnectionConfiguration object used with the DB2ConnectionFactory. This enables specified number of prepared statements to be cached and reused for each DB2Connection object created using this factory. The prepared statements will get cached when execute() method is called for the first time. Alternatively, prepared statements can also be prepared upfront and cached when the connection object is created. See prepare subsection above for more details.

9. Transactions

By default, the auto commit mode is on. This means all the SQLs after they are executed, COMMIT is performed automatically by the driver. When you have more than one SQL statement to be executed as a single atomic unit, then you must start the transaction boundary on your own. The DB2Connection object provides the following methods for this purpose:

a) `beginTransaction()`
b) `commitTransaction()`
c) `rollbackTransaction()`

When a transaction is started manually using beginTransaction() the auto commit mode is turned off until the commit or rollback transaction method is called.

All the four transaction isolation modes of JDBC are supported: 1) Uncommitted Read, 2) Read Committed, 3) Repeatable Read and 4) Serializable. Transaction isolation can be set using `setTransactionIsolationLevel()` method on DB2Connection object. By default, Read Committed isolation level is used. Below is a code example.

```
/**
 * Updates two rows in DEMO table as a transaction.
 *
 * @param src - account id
 * @param amount - tranfer ammount, BigDecimal
 * @param dest - account id
 * @return Mono<Result> - insert return value
 */
public Mono<Void> transferFunds(int src, BigDecimal amount, int dest)
{
    _logger.debug("transferFunds() called, src: "+src+", amount: "+amount+
                        ", dest: "+dest);

    AtomicReference<DB2Connection> con = new AtomicReference<DB2Connection>();

    return Mono.<Void>create(sink -> {
          _pool.getConnection()
                .doOnNext(c -> {
                      con.set(c);
                      c.beginTransaction();
                })
                .flatMap(c -> c.createStatement("UPDATE PHASE1_DEMO “+
                      ”SET ACC_BAL = ACC_BAL - ?, AS_OF = ? WHERE ID = ?")
                            .bind(1, amount)
                            .bind(2, Timestamp.from(Instant.now()))
                            .bind(3, src)
                            .execute())
                .flatMap(r -> r.getNumRowsUpdated()!= 1? Mono.error(
                            new Exception("Debit Failed")) : Mono.just(r))
                .flatMap(r -> con.get().createStatement("UPDATE PHASE1_DEMO “+
                            ”SET ACC_BAL = ACC_BAL + ?, AS_OF = ? WHERE ID = ?")
                               .bind(1, amount)
                               .bind(2, Timestamp.from(Instant.now()))
                               .bind(3, dest)
                               .execute())
                .flatMap(r -> r.getNumRowsUpdated()!= 1? Mono.error(
                            new Exception("Credit Failed")) : Mono.just(r))
                .doOnSuccess(r -> {
                      con.get()
                         .commitTransaction()
                         .doOnSuccess(s -> {
                               con.get().release();
                               _logger.debug("Updated records");
                               sink.success();
                         })
                         .doOnError(e -> {
                               con.get().release();
                               _logger.debug("Commit failed: "+e);
                               sink.error(e);
```

```
                        })
                        .subscribe();
                })
                .doOnError(e -> {
                        _logger.error("Failed to update records: "+e);
                        con.get()
                        .rollbackTransaction()
                        .doOnSuccess(s -> {
                                con.get().release();
                                sink.error(e);
                        })
                        .doOnError(e2 -> {
                                con.get().release();
                                _logger.error("Rollback Failed. "+e2);
                                sink.error(e2);
                        })
                        .subscribe();
                })
                .subscribe();
        });
    }
```

10. Supported Data Types

Below is a figure taken from DB2 for z/OS SQL Reference manual. It lists all the built-in data types supported by DB2. The data types supported by the Reactive Driver are shown by the green boxes around them.

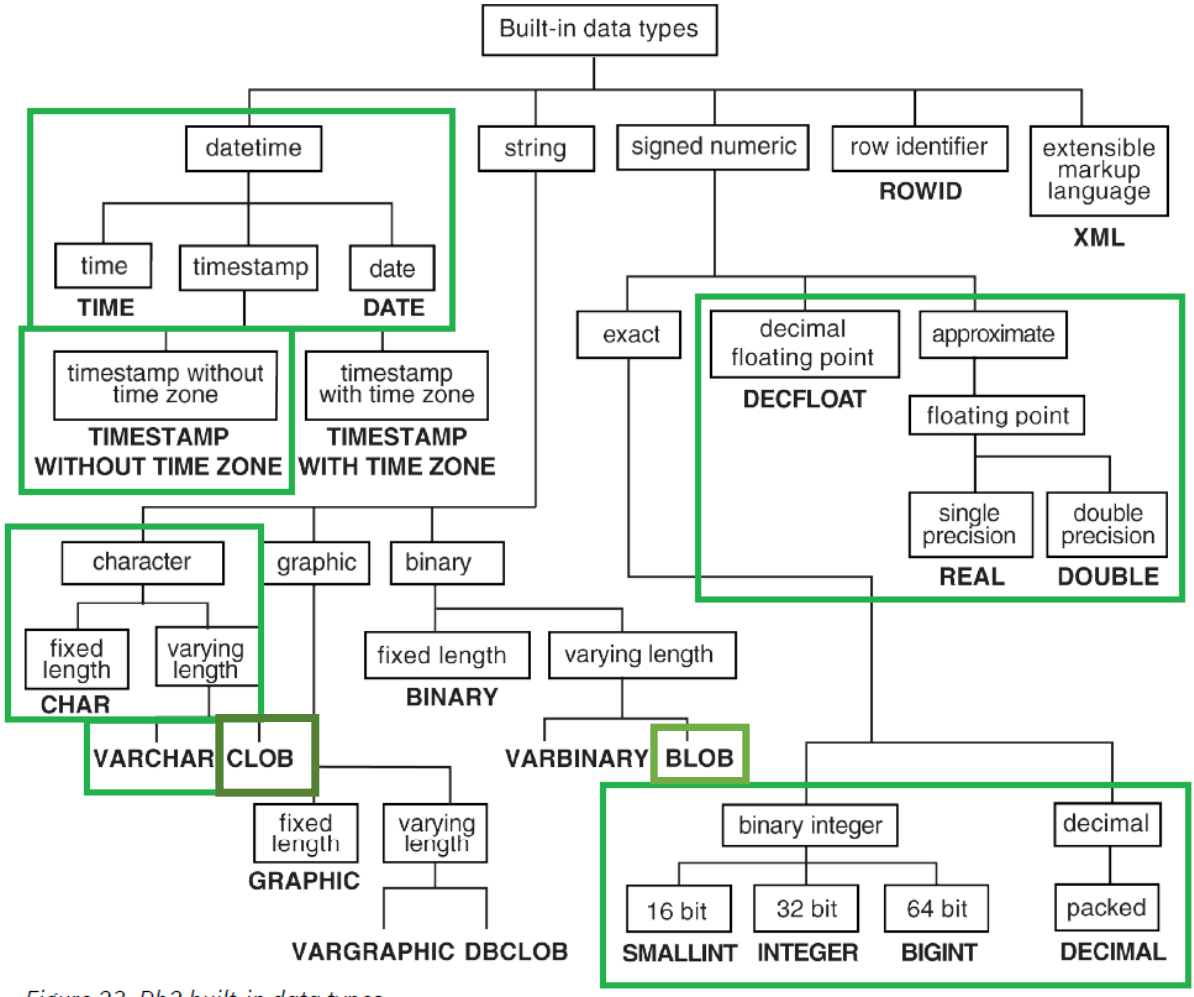

Figure 23. Db2 built-in data types

11. Data Type Mapping

The following table summarizes all the supported DB2 Data Types and the Java Classes that they are mapped to.

DB2 Data Type	Mapped Java Class
SMALLINT	java.lang.Short
INTEGER	java.lang.Integer
BIGINT	java.lang.Long
DECIMAL / NUMERIC	java.math.BigDecimal
DECFLOAT	java.math.BigDecimal
REAL	java.lang.Float
DOUBLE	java.lang.Double
CHAR	java.lang.String
VARCHAR	java.lang.String
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	com.ibm.db2.r2dbc.DB2Blob
CLOB	com.ibm.db2.r2dbc.DB2Clob

- End-of-File