K+apsule (KDB+ Encapsulation Layer)

User/Development Guide

Author: Prasad Polukonda

Document Purpose

This document details the internals of K+apsule which would be helpful to integrate Java applications with KDB+.

Trade Dimensions Technology

© 2011 Trade Dimensions Pvt Ltd. Information contained herein is for Internal Use and may be used only for business purposes authorised by Trade Dimensions Pvt Ltd.

Contents Page

DEVELOPMENT GUIDE **1**

Document Purpose *1*

This document details the internals of K+apsule which would be helpful to integrate Java applications with KDB+. *1*

CONTENTS PAGE **1**

DOCUMENT CONTROL **2**

1 OVERVIEW **3**

1.1 KEY FEATURES 3

2 UML REPRESENTATION & CODE SNIPPETS **4**

2.1 KDBCONNECTION 4

2.1.1 Code snippet: *5*

2.2 KDBACCESSOR 5

2.2.1 Code snippet: *6*

2.3 KDBRECORDTRANSFORMER 6

3 OBJECT - RELATIONAL MAPPING (ORM) **6**

3.1 KDB+ TYPES OF JAVA TYPES 7

3.2 CREATE SCHEMA FROM POJO DEFINITION 8

3.3 MAPPING POJO TO KDB TABLE 8

3.4 MAPPING POJO FIELDS TO KDB TABLE COLUMNS 8

3.5 COLUMN ORDERING 9

3.6 USING ARRAYS/LISTS 10

3.6.1 Mapping array/list against a single column *10*

3.6.2 Mapping array/list to multiple columns *12*

3.7 ENUMS 13

3.8 DATE/TIME 13

3.8.1 Code Snippet *16*

3.9 PAGINATION 16

3.10 HANDLING NULLS 18

3.11 BULK/BATCH INSERTS 18

3.11.1 Batch Properties *19*

3.12 CONFIG BASED ORM 19

3.12.1 Property Config *20*

3.13 AUTO TIME COLUMN SUPPORT 20

4 TOOLS **22**

4.1 KDB POJO GENERATOR: 22

5 FAQ **23**

5.1 WHAT IS THE DEFAULT TIME ZONE THAT THE JAVA INTERFACE TO KDB+ USES? 23

5.2 WHAT IS THE DEFAULT DATE FORMAT? 23

5.3 HOW DOES K+APSULE INTERACT WITH KDB+ SERVER? 24

Document Control

a. Document Ownership

b. Document History

The following table records information regarding released versions of this document and briefly describes changes between versions.

Version	Date	Author	Comment / Changes from Prior Version
1.0	05/07/2012	Prasad Polukonda	Initial Version
1.1	12/08/2012	Prasad Polukonda	Now supports bulk inserts, config based ORM
1.2	18/01/2013	Prasad Polukonda	Now supports the feature that automatically adds a time field when saving the object

1 Overview

K+apsule is a simple API for accessing KDB+ that provides a framework for mapping Java objects to KDB records and vice versa with minimal performance overhead. This API abstracts the complexity in interacting with KDB+ and query generation by providing a simple unified interface. It works same as the way object-relational mapping (ORM) libraries (for eg: Hibernate) work. This API generates the KDB calls and attempts to relieve the developer from manual result set handling and object conversion and keep the application portable to KDB+ with minimal performance overhead.

1.1 Key Features

- Make an entry point to KDB+ and abstracts the layer of complexity and query generation.

- Mapping Java objects (POJO) to KDB records and vice versa. The idea is to make this mapping logic very simple and fast by applying dynamic mapping class generation technique.

- **Developer friendly:** This is one of the main benefits. Developer doesn't need to know syntax of the KDB+ Q operations and would be able to easily integrate any real-time application with KDB with minimal development effort. It minimizes the level of interaction layers and provides a simple unified interface which contains many ease of use operations. No property level java annotations are required if the bean properties are in line with underlying KDB columns. The same applies to the KDB table name. However there will be a facility to override mapping values by using annotations.

- **Support for Arrays/Lists:** There will be a situation where we don't want to provide each single field in a POJO corresponding to a kdb column. For example: you can declare an array as double bid[size] for columns like Bid0, Bid1, Bid2, Bid3…, that automatically maps the corresponding columns based on the array index. The behaviour would be same as the way MkvWrapped supports Arrays.

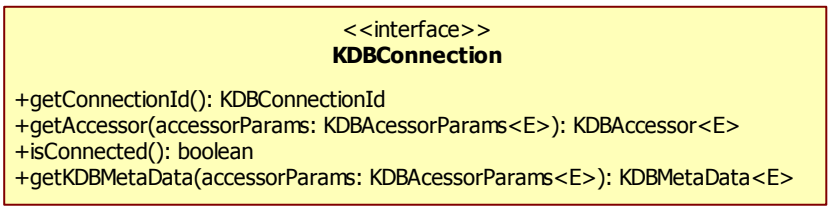
- **Supports KDB ResultSet Pagination**. This feature is useful when you hit historical db that results a large number of records in result set. When handling the large number of records, the client application might end up terminating (or throwing OutOfMemory errors) due to lack enough heap size. K+apsule lets you define a Criteria around the query you are trying to execute and provides the iterable/fixed result set that transparently serves the records based on the window size you set in the criteria.

- Supports JODA DateTime, java.util.Date, java.sql.Date, java.sql.Timestamp: Automatically maps KDB compatible types
- Support for Enums: Automatically maps Java enums to a kdb column.
- Support for executing raw kdb queries
- Support for connecting to multiple KDB Q instances at the same time.
- **Config based ORM (by defining KapsuleORM.xml):** You can use this as an alternative to KDBColumn annotations or for overriding the annotations.
- **Bulk/Batch Inserts:** Bulk inserts can be used to maximize performance. kdb+/tick can handle up to 1 million singleton inserts per second, but many more records per second if the inserts are sent in bulk. Inserting records in bulk can improve performance greatly.

2 UML Representation & Code Snippets

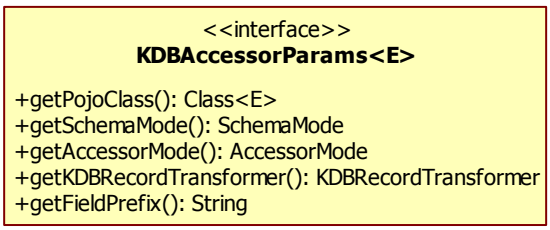
2.1 KDBConnection

This is the entry point to KDB+. An instance of this interface establishes the connection with the KDB+.

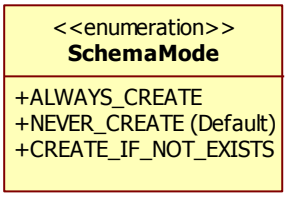


The method *getAccessor(accessorParams: KDBAccessorParams<E>): KDBAccessor<E>* returns a KDBAccessor instance which will be used to execute the queries of a specific POJO type. If the passed-in params doesn't contain KDBRecordTransformer, the instance of KDBAccessor uses default implementation (generates dynamically) of KDBRecordTransformer for mapping POJOs to KDB records and vice versa.

KDBAccessor class encapsulates the properties important for an accessor to use.



Note: Table Schema will be created based on SchemaMode enum type

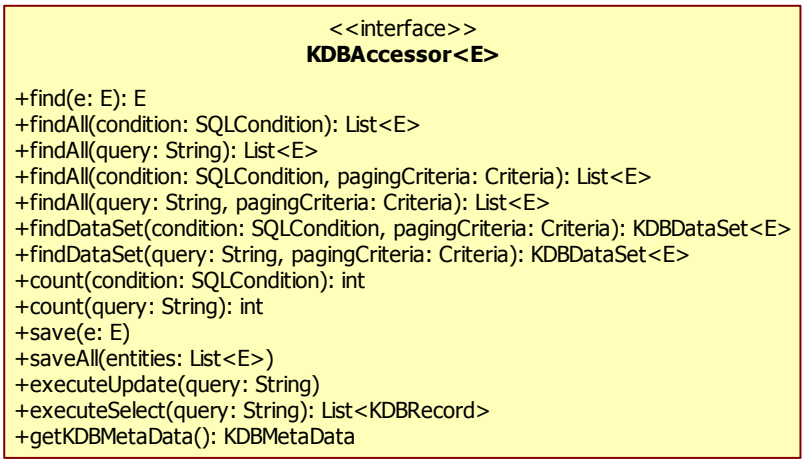


2.1.1 Code snippet:

```
// create a connection Id with host#port#userId:password
KDBConnectionId connectionId = new KDBConnectionId("fikdbap1d", 7500,
"******");

// create a connection object
KDBConnection connection = new DefaultKDBConnection(connectionId);

if (connection.isConnected()) {
    System.out.println("Connected to the " + connectionId);
}
```

2.2 KDBAccessor

This interface provides a layer with some ease of use operations for interacting with KDB+. These operations include fetching the results in the form of Java objects and saving the Java objects into KDB+. It also exposes methods like executeUpdate(), executeSelect() which will enable the user to execute raw queries. There will be one instance of KDBAccessor for a given POJO type.

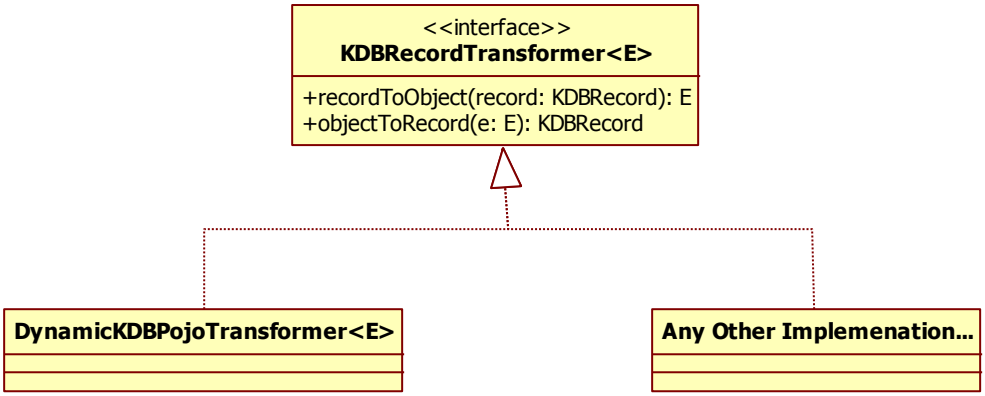


This class dynamically generates (uses the same if exists already) dynamic class implementations of KDBRecordTransformer which will be used to transform KDB record to POJO and vice versa

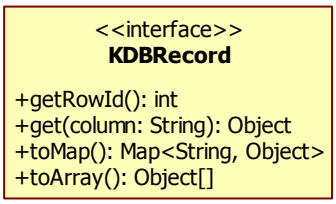
2.2.1 Code snippet:

```
// create a connection object as shown in 2.1.1
…
// crete an accessor on the POJO Type: MyPojo
KDBAccessor<MyPojo>     kdbAccessor     =     connection.getAccessor(new
KDBAccessorParams<MyPojo>(MyPojo.class,  SchemaMode.ALWAYS_CREATE,  null,
AccessorMode.ALL));

// to save an instance of MyPojo
kdbAccessor.save(new MyPojo("K+apsule", "Version 1.0"));

// to retrive the saved instances of MyPojo
List<MyPojo> myPojoList = accessor.findAll(new DefaultSQLCondition());
```

2.3 KDBRecordTransformer



- Where KDBRecord represent raw KDB query results.



3 Object - Relational Mapping (ORM)

K+apsule simplified the process of mapping your POJOs to KDB tables with minimal effort. Using annotations in K+apsule is optional. If the POJO field names and KDB column names are same, you can leave the class without any annotations. K+apsule will also provide you the flexibility to create table schema from your application POJO definition.

Example POJO:

```
public class Price {
```

```
    private String instrumentId;
    private double price;
    /**
     * @return the instrumentId
     */
    public String getInstrumentId() {
        return instrumentId;
    }
    /**
     * @param instrumentId the instrumentId to set
     */
    public void setInstrumentId(String instrumentId) {
        this.instrumentId = instrumentId;
    }
    /**
     * @return the price
     */
    public double getPrice() {
        return price;
    }
    /**
     * @param price the price to set
     */
    public void setPrice(double price) {
        this.price = price;
    }
}
```

3.1 KDB+ Types of Java Types

For reference, internally, types are mapped as follows:

Java Type	Kdb+ Type
boolean	boolean
byte	byte
short	short
integer	int
long	long
float	float
double	float
char	char
String	symbol
Month	month
java.util.Date	date
Time	time
Date	date
Minute	minute
Second	second
Joda Date	date

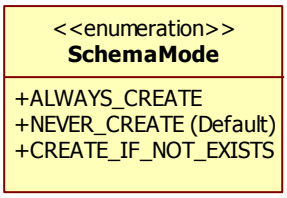
Joda LocalTime	dime
Java.sql.Timestamp	datetime

3.2 Create schema from POJO definition

Again this is optional. K+apsule will never create schema unless you instruct it to do so.
You can instruct K+apsule to do by supplying an optional parameter called *SchemaMode* in your *KDBAccessorParams* object as shown below.

```
KDBAccessor<Price> kdbAccessor = connection.getAccessor(new
KDBAccessorParams<Price>(Price.class,
                SchemaMode.CREATE_IF_NOT_EXISTS, null, AccessorMode.ALL));
```

-



ALWAYS_CREATE: Will create (drops and re-create if one exists) the schema when your application is preparing to start or initialising.

CREATE_IF_NOT_EXISTS: Creates the schema only if table doesn't exist

NEVER_CREATE (default): This is the default mode. It will never try to create the schema and loads the existing schema from the KDB connection.

3.3 Mapping POJO to KDB Table

By default simple name of POJO will be taken as the table. In the above example the class: Price will be mapped to a table 'Price'. If the names are not same, then you can use the annotation @KDBTable to map to the correct table name.

For eg:

```
@KDBTable(name = "BPAPrice", key = @CompositeKey(columns = {}))
public class Price {
…
…
}
```

3.4 Mapping POJO fields to KDB Table Columns

As mentioned before, if POJO field names and KDB column names are same, you can leave the class without tagging them with any annotations. However if the names are different,

you can use @KDBColumn annotation. You can add this annotation at method (getter) or field level.

Note: Using this annotation gives more flexibility for the sub-classes to override the annotation with a different column name

Eg:

```
public class Price {

    private String instrumentId;
    private double price;
    /**
     * @return the instrumentId
     */
    @KDBColumn(name = "Isin")
    public String getInstrumentId() {
        return instrumentId;
    }
    /**
     * @param instrumentId the instrumentId to set
     */
    public void setInstrumentId(String instrumentId) {
        this.instrumentId = instrumentId;
    }
    /**
     * @return the price
     */
    @KDBColumn(name = "MidPrice")
    public double getPrice() {
        return price;
    }
    /**
     * @param price the price to set
     */
    public void setPrice(double price) {
        this.price = price;
    }
}
```

The corresponding KDB table looks like below:

```
Instrument   MidPrice
---------------------
IT0123456789   100.1
IT0123456799   99.78
```

3.5 Column ordering

This is important when instructing K+apsule to create the schema for your POJO. K+apsule can pick up the order of fields of a POJO using the reflection mechanism. This order of the elements differs from one JVM version to other JVM version. In most of the cases, it picks up in alphabetical order. However, at some point of time we have to control this behaviour when we want to keep some particular set of columns on the top of the order always.

This can be done by supplying an optional attribute called 'columnOrder' in @KDBColumn annotation as shown in the below example:

```
public class Price {

    private String instrumentId;
    private double price;
    /**
     * @return the instrumentId
     */
    @KDBColumn(name = "Isin", columnOrder = 0)
    public String getInstrumentId() {
        return instrumentId;
    }
    /**
     * @param instrumentId the instrumentId to set
     */
    public void setInstrumentId(String instrumentId) {
        this.instrumentId = instrumentId;
    }
    /**
     * @return the price
     */
    @KDBColumn(name = "MidPrice", columnOrder = 1)
    public double getPrice() {
        return price;
    }
    /**
     * @param price the price to set
     */
    public void setPrice(double price) {
        this.price = price;
    }
}
```

3.6 Using Arrays/Lists

K+apsule supports array/list (for java primitives, String, Enumeration constants, date, time) fields which cab be either mapped to a single column or spread across multiple columns. By default it will be mapped to a single kdb+ list column.

3.6.1 Mapping array/list against a single column

By default any array or list will be mapped to single column. See the below pojo definition of Depth:

```
public class Depth {

    @KDBColumn(columnOrder = 0)
    private String instrumentId;

    private double[] bid;
    private double[] ask;
    private double[] bidSize;
    private double[] askSize;
```

```
    /**
     * @return the instrumentId
     */
    public String getInstrumentId() {
        return instrumentId;
    }

    /**
     * @param instrumentId
     *            the instrumentId to set
     */
    public void setInstrumentId(String instrumentId) {
        this.instrumentId = instrumentId;
    }

    /**
     * @return the bid
     */
    public double[] getBid() {
        return bid;
    }

    /**
     * @param bid
     *            the bid to set
     */
    public void setBid(double[] bid) {
        this.bid = bid;
    }

    /**
     * @return the ask
     */
    public double[] getAsk() {
        return ask;
    }

    /**
     * @param ask
     *            the ask to set
     */
    public void setAsk(double[] ask) {
        this.ask = ask;
    }

    /**
     * @return the bidSize
     */
    public double[] getBidSize() {
        return bidSize;
    }

    /**
     * @param bidSize
     *            the bidSize to set
     */
    public void setBidSize(double[] bidSize) {
        this.bidSize = bidSize;
    }

    /**
```

```
     * @return the askSize
     */
    public double[] getAskSize() {
        return askSize;
    }

    /**
     * @param askSize
     *            the askSize to set
     */
    public void setAskSize(double[] askSize) {
        this.askSize = askSize;
    }

}
```

The corresponding KDB+ table looks like below:

```
instrumentId ask                    askSize  bid             bidSize
-------------------------------------------------------------------------
IT0123456789 100.05 100.07 100.09 10 10 10 100 99.99 99.98 10 10 10
IT0123456786 100.05 100.07 100.09 10 10 10 100 99.99 99.98 10 10 10
```

3.6.2 Mapping array/list to multiple columns

Alternatively you can map and array or list field to multiple columns in a KDB+ table by using an optional attribute of @KDBColumn annotation as below:

```
@KDBColumn(mapToMultipleColumns = true, size = 3)
```

Where size is the array/list size. This is mandatory field when the mapToMultipleColumn set to true. This is because when you are mapping to columns you should be aware of how many columns you are targeting to.

```
public class Depth {

    @KDBColumn(columnOrder = 0)
    private String instrumentId;

    @KDBColumn(mapToMultipleColumns = true, size = 3)
    private double[] bid;

    @KDBColumn(mapToMultipleColumns = true, size = 3)
    private double[] ask;

    @KDBColumn(mapToMultipleColumns = true, size = 3)
    private double[] bidSize;

    @KDBColumn(mapToMultipleColumns = true, size = 3)
    private double[] askSize;

    + // getters
    …

    + // setters
    …
```

```
}
```

```
instrumentId ask0    ask1    ask2    askSize0 askSize1 askSize2 bid0 bid1
bid2  bidSize0 bidSize1 bidSize2
-------------------------------------------------------------------------
-----------------------------
IT0123456789 100.05 100.07 100.09 10        10        10        100   99.99
99.98 10       10       10
```

3.7 Enums

K+apsule maps Java enum fields to KDB symbols. See below POJO definition of Trade which contains Verb enum.

```
public class Trade {

    private String tradeId;
    private String isin;
    private Verb verb; // Verb is a enum
    private double price;
    private double qty;

    + // getters
    …
    + // setters
    …

}
```

```
public enum Verb {
    Buy, Sell
}
```

Note: It also supports arrays/list of enums.

3.8 Date/Time

K+apsule supports various Date/Time classes For eg: JODA Date, LocalTime, java.util.Date, java.sql.TimeStamp etc. along with KDB Date, Time class types.

The below example demonstrates the usage of Date/Time fields:

```
import java.sql.Timestamp;

import org.joda.time.DateTime;
import org.joda.time.LocalTime;

import com.tdm.kapsule.annotations.KDBColumn;
import com.tdm.kapsule.connection.C.Date;
import com.tdm.kapsule.connection.C.Time;
import com.tdm.kapsule.util.KDBDataType;

public class DateTimeTable {
```

```
    private DateTime jodaDate;
    private LocalTime jodaTime;

    private java.util.Date utilDate;

    private Date kdbDate;
    private Time kdbTime;

    private Timestamp sqlTimeStamp;

    @KDBColumn(type = KDBDataType.DATE_TIME)
    private DateTime jodaTimeStamp;

    /**
     *
     */
    public DateTimeTable() {
        super();
    }

    /**
     * @return the utilDate
     */
    public java.util.Date getUtilDate() {
        return utilDate;
    }

    /**
     * @param utilDate
     *            the utilDate to set
     */
    public void setUtilDate(java.util.Date utilDate) {
        this.utilDate = utilDate;
    }

    /**
     * @return the jodaDate
     */
    public DateTime getJodaDate() {
        return jodaDate;
    }

    /**
     * @param jodaDate
     *            the jodaDate to set
     */
    public void setJodaDate(DateTime jodaDate) {
        this.jodaDate = jodaDate;
    }

    /**
     * @return the jodaTime
     */
    public LocalTime getJodaTime() {
        return jodaTime;
    }

    /**
     * @param jodaTime
     *            the jodaTime to set
     */
```

```
    public void setJodaTime(LocalTime jodaTime) {
        this.jodaTime = jodaTime;
    }

    /**
     * @return the kdbDate
     */
    public Date getKdbDate() {
        return kdbDate;
    }

    /**
     * @param kdbDate
     *            the kdbDate to set
     */
    public void setKdbDate(Date kdbDate) {
        this.kdbDate = kdbDate;
    }

    /**
     * @return the kdbTime
     */
    public Time getKdbTime() {
        return kdbTime;
    }

    /**
     * @param kdbTime
     *            the kdbTime to set
     */
    public void setKdbTime(Time kdbTime) {
        this.kdbTime = kdbTime;
    }

    /**
     * @return the sqlTimeStamp
     */
    public Timestamp getSqlTimeStamp() {
        return sqlTimeStamp;
    }

    /**
     * @param sqlTimeStamp
     *            the sqlTimeStamp to set
     */
    public void setSqlTimeStamp(Timestamp sqlTimeStamp) {
        this.sqlTimeStamp = sqlTimeStamp;
    }

    /**
     * @return the jodaTimeStamp
     */
    public DateTime getJodaTimeStamp() {
        return jodaTimeStamp;
    }

    /**
     * @param jodaTimeStamp
     *            the jodaTimeStamp to set
     */
    public void setJodaTimeStamp(DateTime jodaTimeStamp) {
```

```
        this.jodaTimeStamp = jodaTimeStamp;
    }

}
```

3.8.1 Code Snippet

```
KDBAccessor<DateTimeTable> kdbAccessor = connection.getAccessor(new
KDBAccessorParams<DateTimeTable>(DateTimeTable.class,
                SchemaMode.ALWAYS_CREATE, null, AccessorMode.ALL));

        // Use java.util.Date
        DateTimeTable instance = new DateTimeTable();
        java.util.Date date = new java.util.Date();
        DateTime jodaDateTime = new DateTime(date.getTime());

        instance.setUtilDate(date);
        instance.setJodaDate(jodaDateTime);
        instance.setJodaTime(jodaDateTime.toLocalTime());
        instance.setKdbTime(new Time(date.getTime()));
        instance.setKdbDate(new Date(date.getTime()));
        instance.setSqlTimeStamp(new Timestamp(date.getTime()));
        instance.setJodaTimeStamp(jodaDateTime);
        kdbAccessor.save(instance);
```

3.9 Pagination

In K+apsule, Pagination is the process of splitting the result set into multiple discrete data sets. This is necessary when making a select query against historical data bases which can result huge number of records. In this scenario, your application may not be having enough heap size to hold this particular result set and might end up throwing out of memory errors. The smarter way to deal with this situation is paginating the result set so it can return only limited no of records your application can handle.

This can be achieved by creating paging Criteria when querying the KDB result set. KDBAccessor interface provides the following ease of use operations for this purpose:

```
public List<E> findAll(SQLCondition condition, Criteria pagingCriteria)
throws KDBException;

public List<E> findAll(String query, Criteria pagingCriteria) throws
KDBException;

public KDBDataSet<E> findDataSet(SQLCondition condition, Criteria
pagingCriteria) throws KDBException;

public KDBDataSet<E> findDataSet(String query, Criteria pagingCriteria)
throws KDBException;

public int count(SQLCondition condition) throws KDBException;

public int count(String query) throws KDBException;
```

```
public Criteria(int startRow, int windowSize) {
…
}
```

There are two choices available in overall. Either you can get a fixed result set or iterable data set based your on your paging criteria.

The following method returns list with fixed number of records based on window size you supplied in the criteria object:

```
public List<E> findAll(SQLCondition condition, Criteria pagingCriteria)
throws KDBException;
```

The following method returns an iterable data set which rolls over the query and executes internally till the end of records. The query execution is transparent to the caller:

```
public List<E> findDataSet(SQLCondition condition, Criteria pagingCriteria)
throws KDBException;
```

The following example shows the usage of the above operations:

```
// As the kdb table for this pojo doesn't exist, so use
SchemaMode.ALWAYS_CREATE

KDBAccessor<Quote> kdbAccessor = connection.getAccessor(new
KDBAccessorParams<Quote>(Quote.class,
                SchemaMode.ALWAYS_CREATE, null, AccessorMode.ALL));

// now kdb accessor is ready
// lets do some save operations

kdbAccessor.save(new Quote("IT0123456789", 100.10, 90.80, 10));
kdbAccessor.save(new Quote("IT0123456789", 101.10, 90.80, 10));
kdbAccessor.save(new Quote("IT0123456789", 102.10, 90.80, 10));
kdbAccessor.save(new Quote("IT0123456789", 103.10, 90.80, 10));
kdbAccessor.save(new Quote("IT0123456789", 104.10, 90.80, 10));
kdbAccessor.save(new Quote("IT0123456789", 105.10, 90.80, 10));
kdbAccessor.save(new Quote("IT0123456789", 106.10, 90.80, 10));
kdbAccessor.save(new Quote("IT0123456789", 107.10, 90.80, 10));
kdbAccessor.save(new Quote("IT0123456789", 108.10, 90.80, 10));
kdbAccessor.save(new Quote("IT0123456789", 109.10, 90.80, 10));

// select all
List<Quote> pojoList = kdbAccessor.findAll(new DefaultSQLCondition());
System.out.println("findAll(SQLCondition)=" + pojoList);

int count = kdbAccessor.count(new DefaultSQLCondition());
System.out.println("Total no.of elements saved=" + count);

// select only first 4 rows
pojoList = kdbAccessor.findAll(new DefaultSQLCondition(), new Criteria(0,
3));
System.out.println("findAll(SQLCondition, criteria(0,3)=" + pojoList);

// select rows only from 4-7 rows
pojoList = kdbAccessor.findAll(new DefaultSQLCondition(), new Criteria(4,
```

```
7));
System.out.println("findAll(SQLCondition, criteria(4,7)=" + pojoList);

// Get the iterable data set...it iterates automatically based on the
window size you passed. Here the window size = 3
KDBDataSet<Quote> dataSet = kdbAccessor.findDataSet(new
DefaultSQLCondition(), null);
System.out.println("DataSet:");

// Iterate through the data set
for (Quote quote : dataSet) {
    System.out.println(quote);
}
```

3.10 Handling NULLs

In K+apsule, the Kdb+ null values are mapped to Java values according to the following table:

Kdb+ type	Java null
Boolean	Boolean(false)
Byte	Byte(byte() 0)
Short	Short(Short.MIN_VALUE)
Integer	Integer(Integer.MIN_VALUE)
Long	Long(Long.MIN_VALUE)
Float	Float(Double.NaN)
Double	Double(Double.Nan)
Character	Character(' ')
String	""
Month	Month(Integer.MIN_VALUE)
Date	Data(Integer.MIN_VALUE)
DataTime/java.util.Date	Timestamp(Long.MIN_VALUE)
Minute	Minute(Integer.MIN_VALUE)
Second	Second(Integer.MIN_VALUE)
Time	Time(Integer.MIN_VALUE)

3.11 Bulk/Batch Inserts

Bulk inserts can be used to maximize performance. kdb+/tick can handle up to 1 million singleton inserts per second, but many more records per second if the inserts are sent in bulk. Inserting records in bulk can improve performance greatly. This feature works against only TP (ticker plant) q instance.

K+apsule provides a flexibility to switch between single to bulk insertions and vice versa.
All you need to do is to call: accessor.setBatchEnabled(true) (where accessor is an instance of KDBAccessor).

3.11.1 Batch Properties

The default properties of a batch are: 1000 records, 100ms max waiting time (elapsed time). However you can overwrite the batch parameters with the size you using method: setBatchProperties(BatchProperties batchProperties) on KDBAccessor class.

When batch mode is enabled, it checks whether the number of records received is equal to the batch size you specified in the batch properties or predefined time period is elapsed. If the time constraint has been surpassed, then it only inserts the number of records so far received.

```
KDBConnectionId connectionId = new KDBConnectionId("xxxxxx.eur.nsroot.net",
8020, "kdbadmin", KDBHostType.TP, true);

// create a connection object
KDBConnection connection = new DefaultKDBConnection(connectionId);

KDBAccessor<Quote> kdbAccessor = connection.getAccessor(new
KDBAccessorParams<Quote>(Quote.class, SchemaMode.NEVER_CREATE, null,
AccessorMode.ALL));

// The following setting will enable the batch and will insert 10k at a
//time

kdbAccessor.setBatchProperties(new BatchProperties(10000, 10001));
kdbAccessor.setBatchEnabled(true);
long time = System.nanoTime();

for (int i = 0; i < 100000; i++) {
  kdbAccessor.save(new Quote("IT0123456789", 101.10 + (0.0001 * i), 90.80,
i));
}
```

3.12 Config based ORM

This is an alternative to KDBColumn annotations. You can define ORM mapping in a file called: KapsuleORM.xml. Note that config defined in this file will override the KDB annotations of the class. This applies only for the properties you defined in the xml.

```
<?xml version="1.0" encoding="UTF-8"?>

<kapsule>
<orm class="com.tdm.kdbfeed.pojo.Depth" table="Depth_EUREXMD_LN">
            <property name="id", kdbColumnName="Isin"/>
      </orm>
      <orm class="com.tdm.kdbfeed.pojo.Order" table="Order_EUREX_LN"/>
      <orm class="com.tdm.kdbfeed.pojo.Trade" table="Trade_EUREXMD_LN"/>
</kapsule>
```

In the above configuration, Depth class is mapped to a KDB table called 'Depth_EUREXMD_LN" and the property 'id' is mapped to a column called 'Isin'. If you don't specify the property configuration, K+apsule will take it from the class definition. The following section describes the list of attributes available for a property config.

3.12.1 Property Config

name	Name of the property in the POJO
columnOrder	KDB column order
kdbColumnName	KDB Column Name
ignore	Boolean flag to specify whether this property should be persisted or not
kdbType	KDBType (If the java type and KDB types are not same)
mapToMultipleColumns	This applies to only array/list fields. Set it to “true” if the property needs to be mapped to multiple columns
size	This applies only when the above property is set to true
prefix	Prefix for the array columns
suffix	Suffix for the array columns

3.13 Auto Time column support

This is an additional feature for the POJO definitions that don’t contain time field. This is becoming a problem when people want to use the legacy pojos without making any changes for capturing the time. There is a work around in K+apsule to achieve this:

K+aspsule will automatically create a time object when autoTimeColumn is set to true in your KapsuleORM.xml. KDBTable annotation also contains same property in it.

Note that K+apsule always assume that the ‘time’ column is the first column in the table schema in this case.

```
<?xml version="1.0" encoding="UTF-8"?>

<kapsule>
      <orm class="demo.com.tdm.kapsule.pojo.AutoTimePOJO"
table="AutoTimePOJO" autoTimeColumn="true"/>
</kapsule>
```

Here is an example:

The following POJO definition doesn’t contain time field in it. But K+apsule automatically adds a time field when persisting the objects using above config:

```
public class AutoTimePOJO {

    private String instrumentId;

    private double price;

    /**
     *
     */
    public AutoTimePOJO() {
        super();
    }

    /**
```

```
     * @param instrumentId
     * @param price
     */
    public AutoTimePOJO(String instrumentId, double price) {
        super();
        this.instrumentId = instrumentId;
        this.price = price;
    }

    /**
     * @return the instrumentId
     */
    public String getInstrumentId() {
        return instrumentId;
    }

    /**
     * @param instrumentId
     *            the instrumentId to set
     */
    public void setInstrumentId(String instrumentId) {
        this.instrumentId = instrumentId;
    }

    /**
     * @return the price
     */
    public double getPrice() {
        return price;
    }

    /**
     * @param price
     *            the price to set
     */
    public void setPrice(double price) {
        this.price = price;
    }

    /**
     * Constructs a <code>String</code> with all attributes in name = value
format.
     *
     * @return a <code>String</code> representation of this object.
     */
    @Override
    public String toString() {
        final String DELIM = ",";

        StringBuilder sb = new StringBuilder();
        sb.append(this.getClass().getSimpleName());
        sb.append("(");
        sb.append("instrumentId =
").append(this.instrumentId).append(DELIM);
        sb.append("price = ").append(this.price);
        sb.append(")");
        return sb.toString();
    }

}
```

Results:

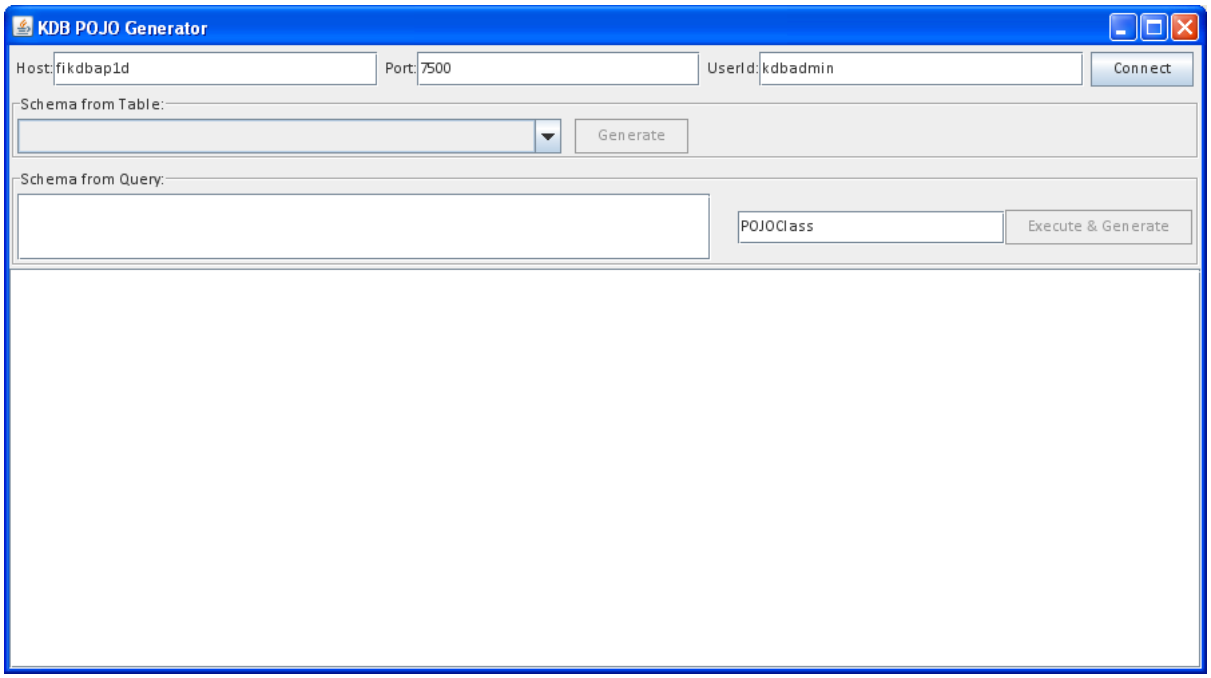
```
time         instrumentId price
------------------------------
12:06:06.103 IT0123456789 100.1
12:06:06.103 IT0123456789 101.1
12:06:06.103 IT0123456789 102.1
12:06:06.103 IT0123456789 103.1
```

4 Tools

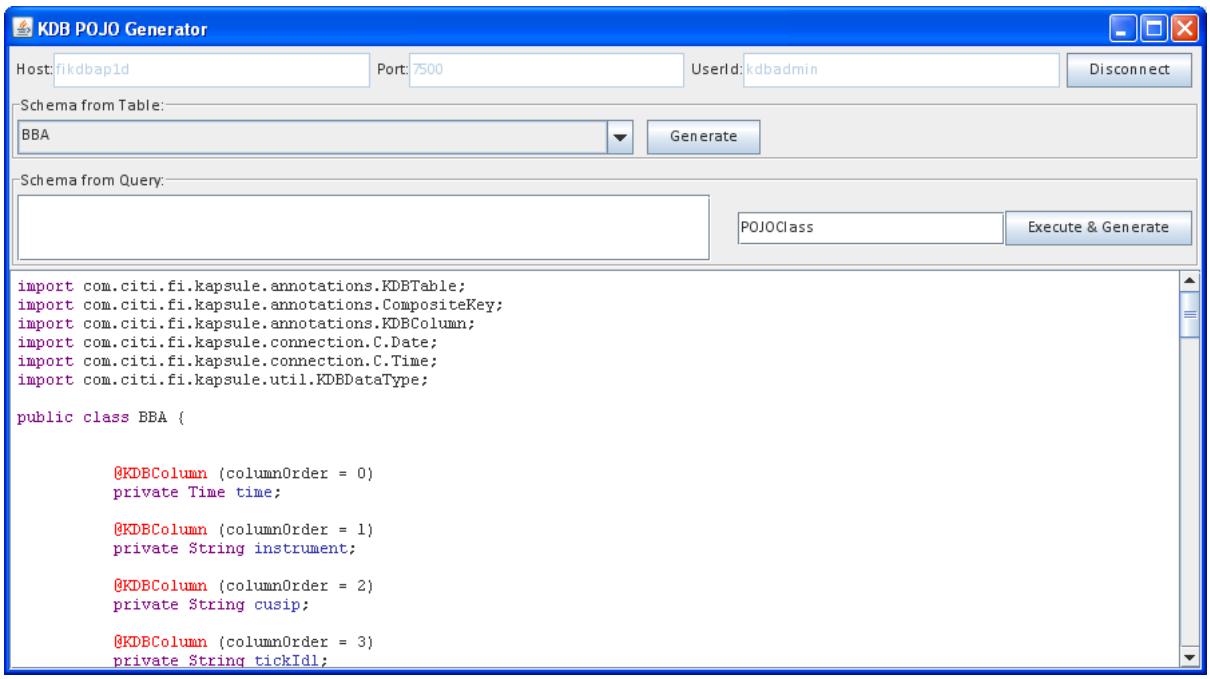
4.1 KDB POJO Generator:

This tool will be available as part of K+apsule library and will be used generate Java POJO for selected KDB tables or by specifying select query. Using this tool, you can simplify the process of creating a POJO based on the table schema or query. You can copy and paste the code from code panel into your Java IDE.

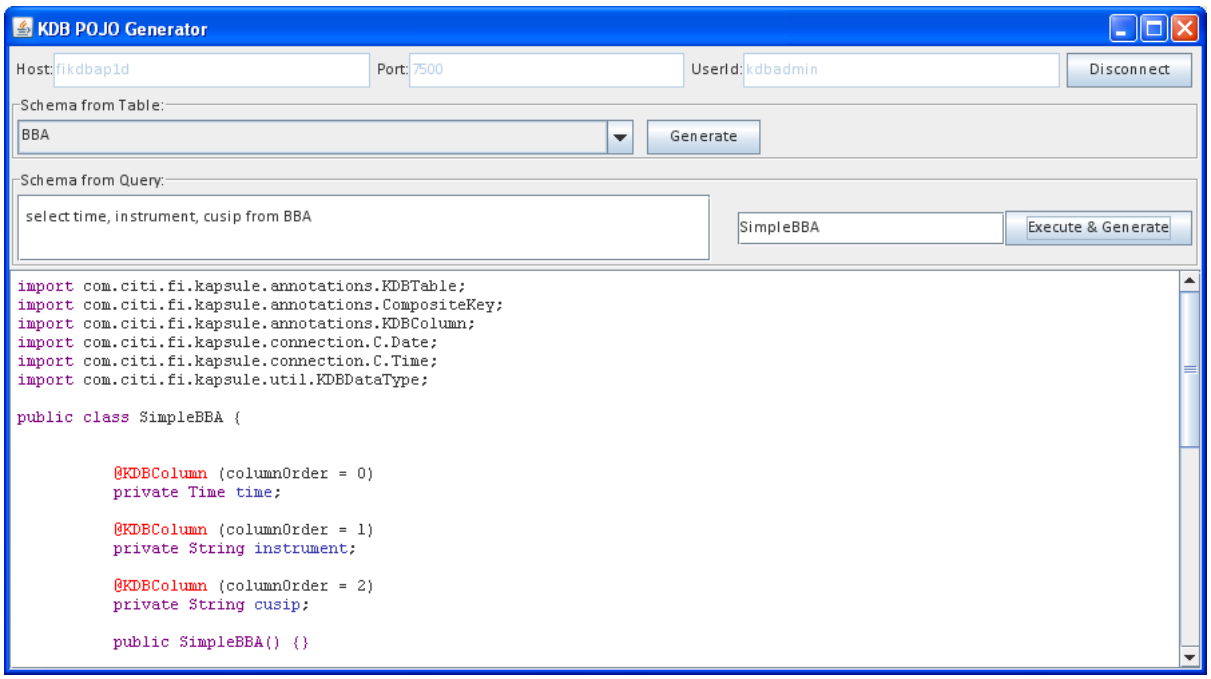
Main-class: com.tdm.kapsule.tools.pojogenerator.KDBPojoGenerator



1) You can connect to any KDB Q instance where you have permissions
2) After connecting to KDB Q instance, you can see list tables in 'Schema from Table'
3) Clicking on 'Generate' will generate a POJO class that matches with the selected table



4) Alternatively, you can generate a POJO from select query as shown below:



5 FAQ

5.1 What is the default time zone that the Java interface to Kdb+ uses?

A) GMT

5.2 What is the default date format?

A) SimpleDateFormat("yyyy.MM.dd")

5.3 How does K+apsule interact with KDB+ server?

A) Although performance is a key consideration, by default K+apsule interacts with KDB+ server in synchronous manner in order to easily throw exceptions in case of any problem. But it is recommended to run feed handlers in asynchronous mode in production. You can control this behaviour on KDBConnectionId class by supplying an optional parameter in the constructor as shown below:

```
KDBConnectionId connectionId = new
KDBConnectionId("firatesln4d.eur.nsroot.net", 8020, "kdbadmin",
KDBHostType.TP, true);
```