JAAD

JAVA API FOR ACTIVEDATABASE

Author: Jawaid Hakim

Contact: mailto:jawaid.hakim@reuters.com

VERSION HISTORY

Version	Comments
Version 0.5	▪ Initial Draft
Version 0.7	▪ Incorporated feedback from Howard Pein and Peter Coates.
Version 1.0	▪ Deleted UML diagrams – the JAVADOCS are much more descriptive.
	▪

REUTERS
CONSULTING

Introduction

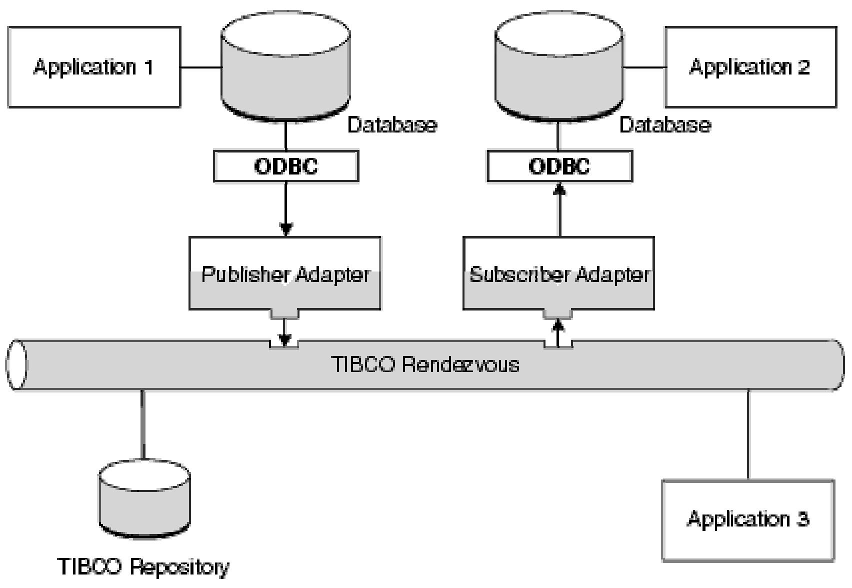
Applications frequently reply on databases to persist critical data. The persistent storage is typically used to store reference data (e.g. user entitlements and offerings) and the application state (e.g. open orders and trades). At the same time, applications will frequently cache some of the data in caches. Caching is usually done to provide quick response times to user requests.

As the state information changes in the cache or the database – for example, trades are generated and orders are filled – the application has to ensure that the two distributed data stores are in a consistent state. When the data cached in the application changes, it is the responsibility of the application to write the new state to persistent storage. At the same time, if data in the persistent storage is modified – e.g. an administrator adds a new user to the system - the application would like to be notified so it can update its in-memory cache.

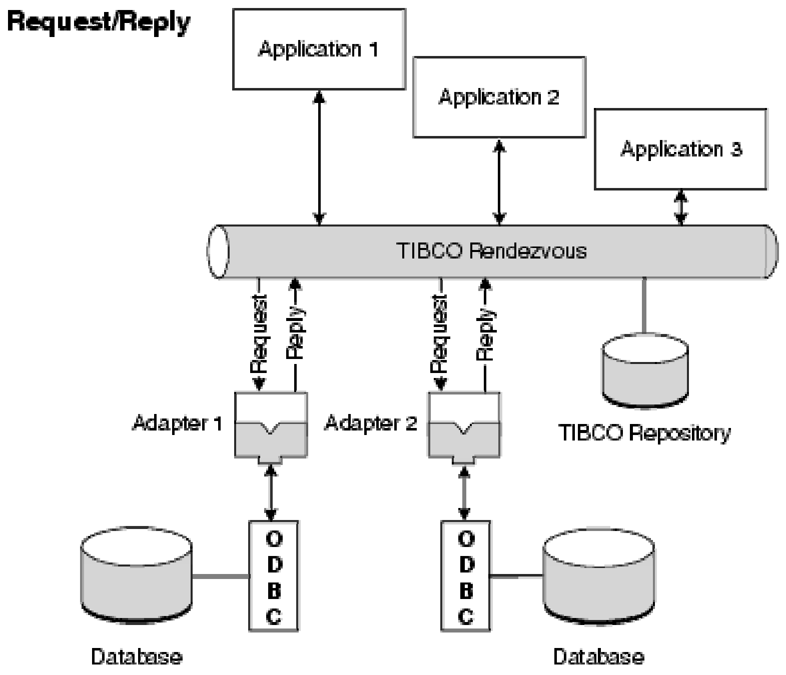
The Adapter for ActiveDatabase (ADB) software is used by applications to *event-enable* databases. ADB publishing agents - in publish/subscribe mode - monitor the database for changes and publish these changes on the TIBCO Rendezvous Bus. As a result, applications are automatically notified of changes to the database as the changes occur.

As shown in the following diagram, when an application updates a table in a database monitored by a publisher adapter, the adapter instance extracts data from the changed rows from database tables and publishes them on appropriate subjects using TIBCO Rendezvous software. The published messages are delivered to subscriber applications efficiently via multicast. A subscriber adapter listening on the corresponding subject receives the messages and updates the relevant tables in its associated database. The data is then available to other applications that have access to the database.

Publish/Subscribe



In addition, applications can run arbitrary SQL queries against a database that is being monitored by ADB agents in the request/reply mode. The result set(s) from the queries are published by the ADB agent on the TIB Bus for consumption by the application. In the request/reply mode, the application publishes a well-formatted message on a subject that the ADB agents have been configured to listen for requests. ADB agents receive this request, execute the SQL against the database, and publish the result set(s) back to the application.



The ADB software specifies the format of the request and reply messages. However, the ADB software does not provide an API for application developers. To send a request to an ADB agent, the application must create a request message with the appropriate fields as defined in the ADB documentation. Similarly, to process a result set, the application must parse the response message.

ADB Request Message Format

A request from an application to an ADB agent must follow the format specified by the ADB software. The following describes the structure of the nested self-describing request message that is sent by an application to the adapter:

```
<Request>
{
rv_Name = "closure", rvmsg_Type = RVMSG_OPAQUE, rvmsg_Data = <optional closure data>
rv_Name = "stmt", rvmsg_Type = RVMSG_RVMSG, rvmsg_Data = <Statement>
rv_Name = "stmt", rvmsg_Type = RVMSG_RVMSG, rvmsg_Data = <Statement>
rv_Name = "stmt", rvmsg_Type = RVMSG_RVMSG, rvmsg_Data = <Statement>
. . .
}
```

where <Statement> is a Rendezvous Message of the following structure:

```
<Statement>
{
rv_Name = "sql", rvmsg_Type = RVMSG_STRING, rvmsg_Data = <The SQL
Statement with possible bind variables>
rv_Name = "maxrows", rvmsg_Type = RVMSG_INT, rvmsg_Data = <Optional:
max number of rows to fetch>
rv_Name = "bind", rvmsg_Type = RVMSG_RVMSG, rvmsg_Data = <Bind data>
rv_Name = "bind", rvmsg_Type = RVMSG_RVMSG, rvmsg_Data = <Bind data>
rv_Name = "bind", rvmsg_Type = RVMSG_RVMSG, rvmsg_Data = <Bind data>
. . .
}
```

where **<Bind data>** is a Rendezvous Message with the following structure:

```
<Bind data>
{
rv_Name = "position", rvmsg_Type = RVMSG_INT, rvmsg_Data = <position of
the placeholder (starting with 1 from left to right)>
rv_Name = "column", rvmsg_Type = RVMSG_STRING, rvmsg_Data = <table-
name.column-name whose column type matches this bound variable>
rv_Name = "data", rvmsg_Type = <type of bound data>, rvmsg_Data =
<value of bound data>
}
```

As one can see from the above message description, it can be somewhat tricky for application developers to get the request format just right.

ADB Response Message Format

A response from an ADB agent also has a well-defined format. The following describes the structure of the nested self-describing response message that is sent by an ADB agent:

```
<Reply>
{
rv_Name = "status" rvmsg_Type = RVMSG_INT rvmsg_Data = 0
rv_Name = "results" rvmsg_Type = RVMSG_RVMSG rvmsg_Data = <Result>
rv_Name = "closure", rvmsg_Type = RVMSG_OPAQUE, rvmsg_Data = <optional
closure data>
}
```

where <Result> is a message in Rendezvous Message format of the following structure:

```
<Result>
{
name = "row" type = RVMSG_RVMSG value = <List of columns>
name = "row" type = RVMSG_RVMSG value = <List of columns>
name = "row" type = RVMSG_RVMSG value = <List of columns>
. . .
}
```

where <List of columns> is a message in Rendezvous Message format of the following structure:

```
<List of columns>
{
```

```
rv_Name = <column-name>, rvmsg_Type = <type of bound data>, rvmsg_Data
= <value of bound data>
rv_Name = <column-name>, rvmsg_Type = <type of bound data>, rvmsg_Data
= <value of bound data>
rv_Name = <column-name>, rvmsg_Type = <type of bound data>, rvmsg_Data
= <value of bound data>
. . .
}
```

If the request processing was not successful, the reply could also return an error code and error description as shown next:

```
<Reply>
{
rv_Name = "status" rvmsg_Type = RVMSG_INT rvmsg_Data = <nonzero number>
rv_Name = "sql" rvmsg_Type = RVMSG_STRING rvmsg_Data = <SQL statement
which caused the error>
rv_Name = "error" rvmsg_Type = RVMSG_STRING rvmsg_Data = <error text>
rv_Name = "closure", rvmsg_Type = RVMSG_OPAQUE, rvmsg_Data = <optional
closure data>
}
```

The status returned is an integer specifying success or an error. Possible values are:

```
0:  ok                 // No error
1:  noMem              // Out of Memory
2:  notInitialized     // Object never initialized
3:  typeConversion     // Type conversion error
4:  dbNotFound         // Database not registered
5:  serverError        // Error reported by server
6:  serverMessage      // Message from server
7:  vendorLib          // Error in vendor's library
8:  notConnected       // Lost connection
9:  endOfFetch         // End of fetch
10: invalidUsage       // invalid usage of object
11: columnNotFound     // Column does not exist
12: invalidPosition    // invalid positioning within
                          object,i.e.bounds err
13: notSupported       // Unsupported feature
14: nullReference      // Null reference parameter
15: notFound           // Database Object not found
16: missing            // Required piece of information is missing
17: noMultiReaders     // This object cannot support multiple
                          readers
18: noDeleter          // This object cannot support deletions
19: noInserter         // This object cannot support insertions
20: noUpdater          // This object cannot support updates
21: noReader           // This object cannot support readers
22: noIndex            // This object cannot support indices
23: noDrop             // This object cannot be dropped
24: wrongConnection    // Incorrect connection was supplied
25: noPrivilege        // This object cannot support privileges
26: noCursor           // This object cannot support cursors
27: cantOpen           // Unable to open
28: applicationError // For errors produced at the application
                          level
29: notReady           // For future use
```

Java API

To address this need for a programmatic interface to the request/reply messages, Reuters Consulting has developed a Java library to enable application developers to easily create ADB requests and to process result rows from the ADB reply. This API shields application developers from the low-level details of the ADB message format. The following sections describe the ADB Java class library.

ADB Package

The ADB Java API package consists of a number of classes. Each class in the package maps to one of the request/reply message constructs specified by the ADB software.

The interesting thing to note is that each class in the *com.reuters.rc.db.adb* implements an interface specified in the *com.reuters.rc.db* package. The *com.reuters.rc.db* package contains a set of interfaces that all concrete implementations – ADB, JDBC, etc. - must implement. In addition, the *com.reuters.rc.db* package provides classes for managing the static – i..e request/reply independent - settings such as the transport, queue, timeout, etc. See Manager for more details.

Request/Reply

In the request/reply mode an application sends a request to ADB agents. The ADB agent processes the request and (optionally) sends a reply message back to the application. The reply message will typically contain one more more result sets generated by executing one or more SQL queries against the database.

ADB Request Statement

A request statement represents a SQL statement or a stored procedure that will be executed by the ADB. The following code shows how a simple request statement is created:

```
try
{
      String sql = …;  // Get the SQL statement

      AdbRequestStmt stmt = new AdbRequestStmt(sql);


}
catch (AdbBusinessException ex)
{
      // Handle exception
}
catch (AdbSystemException ex)
{
      // Handle exception
}
```

Constraining Result Set Size

By default, each request run against the target database can return zero or more result sets. The maximum number of rows returned in a result set is unlimited. Often, it is desirable to limit the maximum number of rows returned in a result set.

The following code limits the maximum number of rows returned from the request statement to 100:

```
try
{
      String sql = …;  // Get the SQL statement

      AdbRequestStmt stmt = new AdbRequestStmt(sql, 100);


}
catch (AdbBusinessException ex)
{
      // Handle exception
}
catch (AdbSystemException ex)
{
      // Handle exception
}
```

ADB Bind

The AdbBind class represents the binding of a data value to a placeholder in the SQL statement. Not every data value sent to an SQL statement must be bound. The client can embed all these values into the actual text of the statement instead. Embedding the values directly into the SQL text results in better performance. However, binary values must be bound.

The following code shows how a request statement is created with one bound value:

```
try
{
      int pos = 1;      // placeholder position
      String col = …    // column name
      Object data = …   // column data

      AdbRequestBind bind = new AdbRequestBind (pos, col, data);

      String sql = …;  // Get the SQL statement

      AdbRequestStmt stmt = new AdbRequestStmt(sql, bind);

}
catch (AdbBusinessException ex)
{
      // Handle exception
}
catch (AdbSystemException ex)
{
      // Handle exception
}
```

The library allows zero or more bind data to be associated with a request statement.

ADB Request

The AdbRequest class is used to create and send a request. The following code shows how a simple request is created and sent to ADB agents:

```
try
{
      String sql = …;  // Get the SQL statement

      AdbRequestStmt stmt = new AdbRequestStmt(sql);

      AdbRequest req = new AdbRequest(stmt);

      String sendSubj = …; // Get the send subject

      // Send the request without waiting for a response
      req.send(sendSubject);

}
catch (AdbBusinessException ex)
{
      // Handle exception
}
catch (AdbSystemException ex)
{
      // Handle exception
}
```

The example shown above is intentionally simple and does not use many of the features provided by the library. In this example, one SQL statement with no bind variable is created and no response is expected. The library fully supports multiple SQL statements with bind variables along with processing of both synchronous and asynchronous responses.

Multiple Request Statements

A single ADB request can contain multiple request (SQL) statements. The following code constructs two request statements within a single ADB request:

```
try
{
      String sql1 = …; // Get the first SQL statement
      String sql2 = …; // Get the second SQL statement

      AdbRequestStmt[] stmts = {new AdbRequestStmt(sql1), new
      AdbRequestStmt(sql2)};

      AdbRequest req = new AdbRequest(stmts);

      String sendSubj = …; // Get the send subject

      // Send the request without waiting for a response
      req.send(sendSubject);

}
catch (AdbBusinessException ex)
```

```
{
      // Handle exception
}
catch (AdbSystemException ex)
{
      // Handle exception
}
```

ADB Reply

Freqently, an application will send a request to an ADB agent and expect a response message in return. The library supports this by providing facilities for both synchronous and asynchronous request/reply.

Synchronous Request/Reply

The following code sends a request and waits for a reply message. The length of time the request waits for a reply is configured through the *DbManager* class. The reply can contain zero of more result sets and each result set can contain zero or more rows.

```
try
{
      String sql = …;  // Get the SQL statement

      AdbRequestStmt stmt = new AdbRequestStmt(sql);

      AdbRequest req = new AdbRequest(stmt);

      String sendSubj = …; // Get the send subject

      // Send the request and wait for response
      AdbReply reply = Req.sendRequest(sendSubject);

      If (reply.isValid())
      {
        // Process each result set – result set index starts at 1
        for (int res = reply.getResultSetCount(); res > 0; --res)
        {
            int rowCount = reply.getResultRowCount(res);

            // Process result set – row index starts at 0
            for (int row = rowCount – 1; row >= 0; --row)
            {
                TibrvMsg row = reply.getResultRow(res, row);

                // Process row
            }
        }
      }
      else
      {
        // Request failed – get the ADB status code and description
        int errCode = reply.getStatus();
        String errDesc = reply.getError();

        // Process error
      }

}
```

```
catch (AdbBusinessException ex)
{
      // Handle exception
}
catch (AdbSystemException ex)
{
      // Handle exception
}
```

Asynchronous Request/Reply

Applications can also send a request to ADB and receive the reply on an application callback. This allows the application to process the reply asynchronously.

The following code sends a request. Instead of waiting for the reply, the application registers an application callback. When the reply is received the application callback is triggered by the library.

```
try
{
      String sql = …;                  // Get the SQL statement

      AdbRequestStmt stmt = new AdbRequestStmt(sql);

      AdbRequest req = new AdbRequest(stmt);

      String sendSubj = …;             // Get the send subject

      String replySubj = …;            // Get the reply subject

      DbRequestCallback cb = …;        // Get the callback

      // Send the request – the reply is received asynchronously by
      // the application callback
      req.sendRequest(sendSubject, cb, replySubj);
}
catch (AdbBusinessException ex)
{
      // Handle exception
}
catch (AdbSystemException ex)
{
      // Handle exception
}

// Reply handler callback
public class AppCallback implements DbRequestCallback
{
      public void onMsg(TibrvListener listener, DbReply reply)
      {
        // Process reply from ADB

        // Close the listener if no more messages are expected
      }
}
```

Publish/Subscribe

ADB publishing agents monitor the database for changes and publish the updates on the TIBCO Bus. The updates can be published either in the TIBCO/Rendezvous or the Minstance data format.

For the TIBCO/Rendezvous format the ADB Reply class described above can be used to parse the data.

The Minstance data format is required if parent/child relationships exist in the database and the child rows are required to be published. In addition, the Minstance format is useful because it includes the name of the source table from which the data is being published. The library provides a set of classes for parsing the Minstance data format messages.

Minstance Message

An Minstance message contains a set of meta-data that provides descriptive information about the published data. This meta-data includes information like the message version, type, etc. The Java API allows the application to easily parse the meta-data and the actual content.

The following code shows how a Minstance messges can be parsed.

```
try
{
        // Get the ADB publication message
        Tibrv pubMsg = …;

        AdbMInstancePubMsg mInstMsg = new AdbMInstancePubMsg(pubMsg);

        // Get the message version
        Integer msgVer = mInstMsg.getMsgVersion();

        // Get the message type
        Integer msgType = mInstMsg.getMsgType();

        // Get the class (table) name
        String tblName = data.getClassName();

        // Get the ADB agent id
        String agentId = data.getAgentId();

        // Get the ADB opcode -INSERT, DELETE, UPDATE, UPSERT
        AdbOpcode opCode = data.getOpcode();

        // Get the sequence number
        Long seqNo = data.getSequence();

        // Get the data - includes all parent/child data
        DbPubMsgData data = mInstMsg.getData();

        // Get the child row data
        DbChildPubMsgData[] childRows = data.getChildRowData();

}
catch (AdbBusinessException ex)
{
      // Handle exception
}
catch (AdbSystemException ex)
{
      // Handle exception
}
```

Manager

The application can configure the library by using static methods of the *DbManager* class. The DbManager class allows the application to specify defaults for the transport (used for publishing request messages), the queue (used for receiving asynchronous reply messages), and the timeout (used for synchronous request/reply messages). Once the defaults have been established, the application does not need to specify the values for these objects. It is also possible for the application to explicitly provide values for the transport, queue, and timeout at the time of making the method call on the relevant classes – these explicit values settings override the defaults.

The following code shows how *DbManager* can be used to configure the library.

```
try
{
        // Get the transport
        TibrvTransport rvTrans = …;

        // Get the queue
        TibrvQueue rvQueue = …;

        // Get the default timeout
        double timeout = …;

        DbManager.setDefaults(rvTrans, rvQueue, timeout);
}
catch (DbBusinessException ex)
{
      // Handle exception
}
catch (DbSystemException ex)
{
      // Handle exception
}
```

Factory

This library was designed to allow applications to customize the persistence strategy without requiring extensive code change. For example, it might be the case that an application requires database access via ActiveDatabase as well as through JDBC.

To allow such flexibility, the library defines a set of interfaces - in the *com.reuters.rc.db* package - that each concrete implementation must provide. For example, classes in the *com.reuters.rc.adb* package implement these interfaces.

Given this framework, a new database access stretegy can be implemented by simply creating a new set of classes that implement the relevant interfaces in the *com.reuters.rc.db* package. Once the new classes have been created, applications can use these new classes with minimal code change.

To further aid this effort, the library defines the *com.reuters.rc.db.DbFactor*y interface. A concrete implementation of this interface is provided by the *com.reuters.rc.AdbFactory* class. If the application uses a factory instance to create all other class instances then switching from one implementation (e.g. ADB) to another (e.g. JDBC) will be as simple as swapping one factory for another.

The following code shows how an application can use the AdbFactory class to create a request:

```
try
{
        // Get the factory
        DbFactory fact = getFactory();

        DbRequestStmt stmt = fact.getRequestStmt(sql);

        DbRequest req = fact.getRequest(stmt);

}
catch (DbBusinessException ex)
{
      // Handle exception
}
catch (DbSystemException ex)
{
      // Handle exception
}


public static DbFactory getFactory()
{
      // Read application configuration and return appropriate factory

      if (config.isAdb())
        return AdbFactory.getInstance();
      else if (config.someOther())
        return SomeOtherFactory.getInstance();
        …
}
```

Conclusion

This Java library provides type-safe access to the ADB request/reply messages. Use of this library should result in significantly less effort for application development teams in using the ADB software.