# Red Hat JBoss Data Virtualization 6.2

# Development Guide Volume 1: Client Development

This guide is for developers wanting to interface to Red Hat JBoss Data Virtualization from within client applications.

# Red Hat JBoss Data Virtualization 6.2 Development Guide Volume 1: Client Development

This guide is for developers wanting to interface to Red Hat JBoss Data Virtualization from within client applications.

Red Hat Customer Content Services

## Legal Notice

## Abstract

This document provides information on concepts and tasks relating to interfacing to Red Hat JBoss Data Virtualization from within client applications.

# Table of Contents

# CHAPTER 1. READ ME

## 1.1. BACK UP YOUR DATA

> **WARNING**
>
> Red Hat recommends that you back up your system settings and data before undertaking any of the configuration tasks mentioned in this book.

## 1.2. VARIABLE NAME: EAP_HOME

**EAP_HOME** refers to the root directory of the Red Hat JBoss Enterprise Application Platform installation on which JBoss Data Virtualization has been deployed.

## 1.3. VARIABLE NAME: MODE

**MODE** will either be **standalone** or **domain** depending on whether JBoss Data Virtualization is running in standalone or domain mode. Substitute one of these whenever you see **MODE** in a file path in this documentation. (You need to set this variable yourself, based on where the product has been installed in your directory structure.)

## 1.4. RED HAT DOCUMENTATION SITE

Red Hat's official documentation site is available at https://access.redhat.com/site/documentation/. There you will find the latest version of every book, including this one.

# CHAPTER 2. CONNECTING TO A VIRTUAL DATABASE

## 2.1. JAVA DATABASE CONNECTIVITY (JDBC)

Java Database Connectivity (JDBC) is an application to database connectivity tool. This Application Program Interface (API) enables communication between applications written in Java and data stored in databases providing methods for data querying and updating.

JDBC is very similar to Open Database Connectivity (ODBC).

## 2.2. JBOSS DATA VIRTUALIZATION AND JDBC

JBoss Data Virtualization provides an API that builds on Java Database Connectivity (JDBC), allowing client applications to issue SQL queries against deployed virtual databases (VDBs).
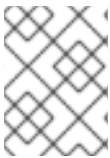
> **NOTE**
>
> The Java 1.6 or 1.7 JDK is required by client applications connecting to an JBoss Data Virtualization VDB.

> **NOTE**
>
> The JBoss Data Virtualization JDBC API is compatible with the JDBC 4.0 specification but does not fully support all methods. Advanced features, such as *updatable* result sets and SQL3 data types are also not supported.
>
> See Section A.3, "Unsupported Classes and Methods in java.sql" and Section A.4, "Unsupported Classes and Methods in javax.sql" for more information about unsupported classes and methods.

## 2.3. GENERATED KEYS

JBoss Data Virtualization supports returning generated keys for JDBC sources and from JBoss Data Virtualization temp tables with SERIAL primary key columns. However the current implementation returns only the last set of keys generated and returns the key results directly from the source - no view projection of other intermediate handling is performed. For most scenarios (single source inserts) this handling is sufficient. A custom solution may need to be developed if you are using a FOR EACH ROW instead of trigger to process your inserts and target multiple tables that each return generated keys. It is possible to develop a UDF that also manipulates the returned generated keys - see the `org.teiid.CommandContext` methods dealing with generated keys for more.

## 2.4. CONNECTION METHODS

The following are methods of creating a JDBC connection to an JBoss Data Virtualization virtual database (VDB):

- Using the `org.teiid.jdbc.TeiidDriver` driver class

- Using the `org.teiid.jdbc.TeiidDataSource` data source class

- Configuring a VDB as a JBoss data source

## 2.5. CONNECTING TO A VDB USING THE DRIVER CLASS

The **org.teiid.jdbc.TeiidDriver** class, found in **EAP_HOME/dataVirtualization/jdbc/teiid-VERSION-redhat-N-jdbc.jar**, should be used to create a connection using the Java **java.sql.DriverManager** class.

## 2.6. CREATE A CONNECTION TO A VDB USING THE DRIVER CLASS

**Prerequisites**

- The client application must have the JBoss Data Virtualization **EAP_HOME/dataVirtualization/jdbc/teiid-VERSION-redhat-N-jdbc.jar** JAR file in its classpath. (If the application is running on the same application server as the JBoss Data Virtualization instance to which the connection is desired, then this will already be the case.)

- JBoss Data Virtualization must be installed and running, with the relevant virtual databases (VDBs) deployed.

**Procedure 2.1. Create a Connection to a VDB Using the Driver Class**

- **Create a Connection to the VDB**
  Within your client application code, create a **Connection** to the VDB using the **DriverManager.getConnection()** method:

  ```
  public class TeiidClient
  {
      public Connection getConnection(String user, String password)
  throws Exception
      {
          String url =
  "jdbc:teiid:myVDB@mm://localhost:31000;ApplicationName=myApp";
          return DriverManager.getConnection(url, user, password);
      }
  }
  ```

**See Also:**

- Section 2.7, "Driver Connection URL Format"

## 2.7. DRIVER CONNECTION URL FORMAT

URLs used when establishing a connection using the driver class have the following format:

```
jdbc:teiid:VDB-NAME@mm[s]://HOSTNAME:PORT;[prop-name=prop-value;]*
```

Given this format, the following table describes the variable parts of the URL:

**Table 2.1. URL Entities**

| Variable Name | Description |
| --- | --- |

| Variable Name | Description |
|---|---|
| VDB-NAME | The name of the virtual database (VDB) to which the application is connected.<br><br>**IMPORTANT**<br><br>VDB names can contain version information; for example, **myvdb.2**. If such a name is used in the URL, this has the same effect as supplying a **version=2** connection property. Note that if the VDB name contains version information, you cannot also use the **version** property in the same request. |
| mm[s] | The JBoss Data Virtualization JDBC protocol. **mm** is the default for normal connections. **mms** uses SSL for encryption and is the default for the AdminAPI tools. |
| HOSTNAME | The server where JBoss Data Virtualization is installed. |
| PORT | The port on which JBoss Data Virtualization is listening for incoming JDBC connections. |
| [prop-name=prop-value] | Any number of additional name-value pairs can be supplied in the URL, separated by semi-colons. Property values must be URL encoded if they contain reserved characters, for example, **?**, **=**, and **;** . |

## 2.8. CONNECTION PROPERTIES FOR THE DRIVER AND DATA SOURCE CLASSES

The following table shows all the connection properties that can be used with the JBoss Data Virtualization JDBC driver URL connection string and the JBoss Data Virtualization JDBC data source class.

These properties are defined in **org.teiid.net.TeiidURL** and **org.teiid.jdbc.ExecutionProperties** (**waitForLoad** is defined in **org.teiid.jdbc.EmbeddedProfile**), and the corresponding set methods are defined for the data source class in **org.teiid.jdbc.TeiidDataSource** and its superclass, **org.teiid.jdbc.BaseDataSource**.

Property names that can be used in the driver URL connection string are listed in the *Property Name* column, and the corresponding set methods for use with the data source class are listed in the *Method Name* column.

**Table 2.2. Connection Properties**

| Property Name | Method Name | Type | Description |
|---|---|---|---|
| **ansiQuotedIdentifiers** | **setAnsiQuotedIdentifiers** | **boolean** | Sets the parsing behavior for double quoted entries in SQL. If true, then parses doubled quoted entries as identifiers. If false, then double quoted values that are valid string literals are parsed as string literals. Default is true. |

| Property Name | Method Name | Type | Description |
|---|---|---|---|
| **ApplicationName** | **setApplicationName** | **String** | Name of the client application; allows the administrator to distinguish between connections. |
| **autoCommitTxn** | **setAutoCommitTxn** | **String** | Only applies only when autoCommit is set to "true". This determines how an executed command needs to be wrapped as a transaction inside the JBoss Data Virtualization engine to maintain the data integrity. <ul><li>ON - Always wrap command in distributed transaction</li><li>OFF - Never wrap command in distributed transaction</li><li>DETECT (default)- If the executed command is spanning more than one source it automatically uses distributed transaction.</li></ul> See Section 5.1, "JDBC Transaction Types" for more information. |
| **autoFailover** | **setAutoFailover** | **boolean** | If true, automatically selects a new server instance after a communication exception. This is typically not needed when connections are managed, as the connection can be purged from the pool. Default is false. |
| **disableLocalTxn** | **setDisableLocalTxn** | **boolean** | If true, the autoCommit setting, commit and rollback is ignored for local transactions. Default is false. |
| **fetchSize** | **setFetchSize** | **int** | Size of the resultset. Default is 2048. <=0 indicates that the default should be used. |
| **NOEXEC** | **setNoExec** | **String** | (Typically not set as a connection property.) Can be ON or OFF. ON prevents query execution, but parsing and planning still occurs. Default is OFF. |
| **partialResultsMode** | **setPartialResultsMode** | **boolean** | Enable/disable support partial results mode. Default false. See Section 4.12, "Partial Results Mode" for more details. |

| Property Name | Method Name | Type | Description |
|---|---|---|---|
| `Passthrough Authentication` | `setPassthrough Authentication` | **boolean** | Only applies to local connections. When this option is set to true, JBoss Data Virtualization looks for an authenticated security context on the calling thread. If one is found, it uses that user's credentials to create a session. JBoss Data Virtualization also verifies that the same user is using this connection during the life of the connection. If it finds a different security context on the calling thread and the new user is eligible to login to JBoss Data Virtualization, it switches the identity on the connection. |
| `useCallingThread` | Not applicable. Must be set using **`setAdditionalPro perties`** method if connecting via the data source class. | **boolean** | Only applies to local connections. When this option is set to true, the calling thread is used to process the query. If false, then an engine thread is used. Default is true. |
| `QUERYTIMEOUT` | `setQueryTimeout` | **int** | Default query timeout in seconds. Must be >= 0. 0 indicates no timeout. Can be overridden by **`Statement.setQueryTimeout`**. Default is 0. |
| `useJDBC4ColumnNa me AndLabelSemantic s` | `setUseJDBC4Colum nName AndLabelSemantic s` | **boolean** | A change was made in JDBC4 to return unaliased column names as the ResultSetMetadata column name. Prior to this, if a column alias was used it was returned as the column name. Setting this property to false enables backwards compatibility when JDBC3 and older support is still required. Defaults to true. |
| `password` | `setPassword` | **String** | Credential for user |
| `resultSetCacheMo de` | `setResultSetCach eMode` | **boolean** | ResultSet caching is turned on/off. Default is false. |
| `SHOWPLAN` | `setShowPlan` | **String** | (Typically not set as a connection property.) Can be ON, OFF or DEBUG. ON returns the query plan along with the results and DEBUG additionally prints the query planner debug information in the log and returns it with the results. Both the plan and the log are available through JDBC API extensions. Default is OFF. |

| Property Name | Method Name | Type | Description |
| --- | --- | --- | --- |
| `user` | `setUser` | `String` | User name. |
| `version` | `setDatabaseVersion` | `String` | Version number of the VDB. |
| `jaasName` | `setJaasName` | `String` | JAAS configuration name. Only applies when configuring GSS authentication. See the *JBoss Data Virtualization Administration Guide* for more information about GSS configuration. |
| `kerberosServicePrincipleName` | `setKerberosServicePrincipleName` | `String` | Kerberos authenticated principle name. Only applies when configuring GSS authentication. See the *JBoss Data Virtualization Administration Guide* for more information about GSS configuration. |
| `encryptRequests` | `setEncryptRequests` | `boolean` | Only applies to non-SSL socket connections. When set to true, the request message and any associated payload is encrypted using the connection cryptor. Default is false. |
| `waitForLoad` | Not applicable. Needs to be specified using `setAdditionalProperties` if set using the data source class. | `String` | Only applies to local connections. When this option is set to a non-negative value, the connection will wait that number of milliseconds for the VDB to become active. Setting to a negative number uses the system default setting. |

## 2.9. CONNECTING TO A VDB USING THE DATA SOURCE CLASS

The `org.teiid.jdbc.TeiidDataSource` class is based on the `javax.sql.DataSource` connection factory. It can be used to create `ManagedConnection`s and `XAConnection`s to both `DataSource`s and `XADataSource`s. XA transactions are extended to JBoss Data Virtualization sources that support XA.

The JBoss Data Virtualization `DataSource` class is *serializable* and can be used with JNDI naming services.

See *EAP_HOME*`/quickstarts/simpleclient/` example for more information.

## 2.10. CREATE A CONNECTION TO A VDB USING THE DATA SOURCE CLASS

**Prerequisites**

- The client application must have the JBoss Data Virtualization *EAP_HOME*/dataVirtualization/jdbc/teiid-*VERSION*-redhat-*N*-jdbc.jar JAR file in its classpath. (If the application is running on the same application server as the JBoss Data Virtualization instance to which the connection is desired, then this will already be the case.)

- JBoss Data Virtualization must be installed and running, with the relevant virtual databases (VDBs) deployed.

**Procedure 2.2. Create a Connection to a VDB Using the Data Source Class**

- **Create a Connection Object**
  Create a **org.teiid.jdbc.TeiidDataSource** object, set the required properties, and use the **TeiidDataSource.getConnection()** method to obtain a Connection object. For example:

  ```
  public class TeiidClient
  {
      public Connection getConnection(String user, String password)
  throws Exception
      {
          TeiidDataSource ds = new TeiidDataSource();
          ds.setUser(user);
          ds.setPassword(password);
          ds.setServerName("localhost");
          ds.setPortNumber(31000);
          ds.setDatabaseName("myVDB");
          return ds.getConnection();
      }
  }
  ```

**See Also:**

- Section 2.8, "Connection Properties for the Driver and Data Source Classes"

- Section 2.11, "Additional Data Source Connection Properties"

## 2.11. ADDITIONAL DATA SOURCE CONNECTION PROPERTIES

When using the driver class, various properties are derived from the URL. For the data source class, these properties are set using the following additional methods:

**Table 2.3. Data Source Connection Properties**

| Method Name | Type | Description |
|---|---|---|
| **setAlternateServers** | **String** | Optional delimited list of host:port entries. Refer to Section 3.1, "Using Multiple Hosts" for more information. |
| **setAdditionalProperties** | **String** | Optional setting of properties that has the same format as the property string in a driver connection URL. Refer to Section 2.7, "Driver Connection URL Format" |

| Method Name | Type | Description |
| --- | --- | --- |
| **setDatabaseName** | **String** | The name of a virtual database (VDB) deployed to JBoss Data Virtualization.<br><br>**IMPORTANT**<br><br>VDB names can contain version information; for example, **myvdb.2**. If such a name is used in the URL, this has the same effect as supplying a **version=2** connection property. Note that if the VDB name contains version information, you cannot also use the **version** property in the same request. |
| **setDatabaseVersion** | **String** | The VDB version. |
| **setDataSourceName** | **String** | The name given to this data source |
| **setPortNumber** | **int** | The port number on which the server process is listening. |
| **setServerName** | **String** | The server hostname where the JBoss Data Virtualization runtime is installed. |
| **setSecure** | **boolean** | Secure connection. Flag to indicate to use SSL (mms) based connection between client and server. |

**NOTE**

All the properties from URL Connection Properties can be used on DataSource using the AdditionalProperties setter method if the corresponding setter method is not already available. For example, you can add **useCallingThread** property as

```
<xa-datasource-property
name="AdditionalProperties">useCallingThread=false</xa-
datasource-property>
```

## 2.12. CONNECTING TO A VDB AS A DATA SOURCE

JBoss Data Virtualization virtual databases (VDBs) can be configured as an JBoss Enterprise Application Platform (EAP) data source. The data source can then be accessed from JNDI or injected into your Java EE applications. A JBoss Data Virtualization data source is deployed in the same way as any other database resource.

**NOTE**

The recommended approach for configuring data sources is to use JBoss CLI or Management Console, not directly editing the **standalone.xml** configuration file. See the Red Hat JBoss Enterprise Application Platform *Administration and Configuration Guide* for more information on how to configure data sources in JBoss EAP.

# CHAPTER 3. MULTIPLE HOSTS

## 3.1. USING MULTIPLE HOSTS

JBoss Data Virtualization may be clustered over several servers utilizing *failover* and *load balancing*.

The easiest way to enable these features is for the client to specify multiple hostname and port number combinations in the URL connection string as a comma separated list of **host:port** combinations:

```
jdbc:teiid:<vdb-name>@mm://host1:31000,host1:31001,host2:31000;version=2
```

If you are connecting with the data source class, the **setAlternateServers** method can be used to specify the failover servers. The format is also a comma separated list of **host:port** combinations.

The client randomly selects one of the JBoss Data Virtualization servers from the list and establishes a session with that server. If a connection cannot be established, then each of the remaining servers will be tried in random order. This allows for both connection time failover and random server selection load balancing.

## 3.2. FAILOVER

Post connection failover will be used if you are using an administration connection (such as what is used by AdminShell) or if the **autoFailover** connection property is set to true. Post connection failover works by sending a ping, at most every second, to test the connection prior to use. If the ping fails, a new instance will be selected prior to the operation being attempted.

This is not considered to be true transparent application failover because the client does not restart transactions or queries, nor will it recreate session scoped temporary tables.

> ⚠️ **WARNING**
>
> Extreme caution should be exercised if using this with non-admin connections.

## 3.3. LOAD BALANCING

Post connection load balancing can be utilized in one of two ways.

If you are using the **TeiidDataSource** class and the **PooledConnection** returned by **getPooledConnection** is terminated using the **close()** method, then a new server instance will be selected automatically. (When using driver based connections or when using the **TeiidDataSource** class in a connection pool, the automatic load balancing will not happen.)

Alternatively, you can explicitly trigger load balancing through the use of the set statement: **SET NEWINSTANCE TRUE**. Typically you will not issue this statement manually, but you can use it as the connection test query on your data source configuration:

```
<datasources>
        <datasource jndi-name="java:/teiidDS" pool-name="teiidDS">
```

```
            <connection-url>jdbc:teiid:
{vdb}@mm://{host}:31000</connection-url>
            <driver>teiid</driver>
            <pool>
                <prefill>false</prefill>
                <use-strict-min>false</use-strict-min>
                <flush-strategy>FailingConnectionOnly</flush-strategy>
                <check-valid-connection-sql>SET NEWINSTANCE TRUE</check-
valid-connection-sql>
            </pool>
            <security>
                <user-name>{user}</user-name>
                <password>{password}</password>
            </security>
        </datasource>
        <drivers>
            <driver name="teiid" module="org.jboss.teiid.client">
                <driver-class>org.teiid.jdbc.TeiidDriver</driver-class>
                <xa-datasource-class>org.teiid.jdbc.TeiidDataSource</xa-
datasource-class>
            </driver>
        </drivers>
    </datasources>
```
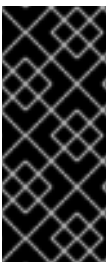
> **IMPORTANT**
>
> Session level temporary tables, currently running transactions, session level cache entries, and PreparedPlans for a given session will not be available on other cluster members. Therefore, it is recommended that post connection time load balancing is only used when the logical connection could have been closed, but the actual connection is reused. This is the typical connection pool pattern.

## 3.4. INCREASE THE MAXIMUM NUMBER OF CACHED INSTANCES

JBoss Data Virtualization maintains a pool of extra socket connections that are reused. For load balancing, this reduces the potential cost of switching a connection to another server instance. The default setting is to maintain 16 connections. If the client application is connecting to a large number of JBoss Data Virtualization instances and post connection time load balancing is used, then consider increasing the number of cached instances.

**Procedure 3.1. Increase the Maximum Number of Cached Instances**

- Increase the **org.teiid.sockets.maxCachedInstances** property in the **teiid-client-settings.properties** file located in *EAP_HOME***/dataVirtualization/jdbc/teiid-***VERSION***-redhat-***N***-jdbc.jar**.

  > **NOTE**
  >
  > An example file (**teiid-client-settings.orig.properties**) is packaged in the client JAR file.

## 3.5. ADVANCED CONFIGURATION

Features such as server discovery, load balancing, failover, retry, and retry delay, may be customized if the default policy is not appropriate. Refer to the **org.teiid.net.socket.ServerDiscovery** interface and default implementation **org.teiid.net.socket.UrlServerDiscovery** for more information on customization.

The **UrlServerDiscovery** implementation provides the following features:

- discovery of servers from URL hosts (including the data source server and alternative servers)

- random selection for load balancing and failover

- one connection attempt per host

- no biasing

- black listing

Typically you'll want to extend the **UrlServerDiscovery** so that it can be used as the fall-back strategy and to only implement the necessary changed methods. It is important to consider that one **ServerDiscovery** instance will be created for each connection. Any sharing of information between instances should be done through static state or some other shared lookup.

Your customized server discovery class must be referenced by the **discoveryStrategy** connection/DataSource property using its full class name.

## 3.6. REAUTHENTICATION

JBoss Data Virtualization connections (defined by the **org.teiid.jdbc.TeiidConnection** interface) support the **changeUser** method to reauthenticate a given connection. If reauthentication is successful, the current connection may be used with the given identity. Existing statements and resultsets are still available for use under the old identity.

# CHAPTER 4. EXTENSIONS TO JDBC

## 4.1. PREPARED STATEMENTS

JBoss Data Virtualization provides **org.teiid.jdbc.TeiidPreparedStatement**, a custom interface for the standard **java.sql.PreparedStatement**, and implementations **org.teiid.jdbc.CallableStatementImpl** and **org.teiid.jdbc.PreparedStatementImpl**. Prepared statements can be important in speeding up common statement execution, since they allow the server to skip parsing, resolving, and planning of the statement.

The following points should be considered when using prepared statements:

- It is not necessary to pool client-side JBoss Data Virtualization prepared statements, because JBoss Data Virtualization performs plan caching on the server side.

- The number of cached plans is configurable. The plans are purged in order of least recently used (LRU).

- Cached plans are not distributed through a cluster. A new plan must be created for each cluster member.

- Plans are cached for the entire VDB or for just a particular session. The scope of a plan is detected automatically based upon the functions evaluated during its planning process.

- Runtime updates of costing information do not yet cause re-planning. At this time only session-scoped temporary table or internally materialized tables update their costing information.

- Stored procedures executed through a callable statement have their plans cached in the same way as a prepared statement.

- Bind variable types in function signatures, for example **where t.col = abs(?)**, can be determined if the function has only one signature or if the function is used in a predicate where the return type can be determined. In more complex situations it may be necessary to add a type hint with a cast or convert, for example **upper(convert(?, string))**.

## 4.2. JDBC STATEMENT EXTENSIONS

The JBoss Data Virtualization statement interface, **org.teiid.jdbc.TeiidStatement**, provides functionality beyond the JDBC standard. To use this interface, cast or unwrap the statement returned by the connection. The interface provides the following methods for setting and retrieving statement properties:

**Table 4.1. Statement Properties**

| Method Name | Description |
|---|---|
| **getAnnotations** | This method has been deprecated. Use the SHOW statement. |
| **getDebugLog** | This method has been deprecated. Use the SHOW statement. |
| **getExecutionProperty** | This method has been deprecated. Use the SHOW statement. |

| Method Name | Description |
|---|---|
| **getPlanDescription** | Get the query plan description if the statement was last executed with SHOWPLAN ON\|DEBUG. The plan is a tree made up of **org.teiid.client.plan.PlanNode** objects. Typically **PlanNode.toString()** or **PlanNode.toXml()** will be used to convert the plan into a textual form. |
| **getRequestIdentifier** | Get an identifier for the last command executed on this statement. If no command has been executed yet, null is returned. |
| **setExecutionProperty** | **NOTE** <br><br> This method has been deprecated. Use the SET statement. |
| **setPayload** | Set a per-command payload to pass to translators. Currently the only built-in use is for sending hints for an Oracle data source. |

## 4.3. NON-BLOCKING STATEMENT EXECUTION

JDBC query execution can indefinitely block the calling thread when a statement is executed or a resultset is being iterated. In some situations you may wish to have your calling threads held in these blocked states. When using embedded connections, you may optionally use the **org.teiid.jdbc.TeiidStatement** and **org.teiid.jdbc.TeiidPreparedStatement** interfaces to execute queries with a callback **org.teiid.jdbc.StatementCallback** that will be notified of statement events, such as an available row, an exception, or completion. Your calling thread will be free to perform other work. The callback will be executed by an engine processing thread as needed. If your results processing is blocking and you want query processing to run concurrently with results processing, then your callback should implement onRow handling in a multi-threaded manner to allow the engine thread to continue.

```
PreparedStatement stmt = connection.prepareStatement(sql);
TeiidPreparedStatement tStmt = stmt.unwrap(TeiidPreparedStatement.class);
tStmt.submitExecute(new StatementCallback() {
    @Override
    public void onRow(Statement s, ResultSet rs) {
            //any logic that accesses the current row ...
        System.out.println(rs.getString(1));
    }

    @Override
    public void onException(Statement s, Exception e) throws Exception {
        s.close();
    }

    @Override
    public void onComplete(Statement s) throws Exception {
```

```
        s.close();
    }, new RequestOptions()
});
```

**NOTE**

The non-blocking logic is limited to statement execution only. Other JDBC operations, such as connection creation or batched executions do not yet have non-blocking options.

If you access forward positions in the onRow method (calling next, isLast, isAfterLast, absolute), they may not yet be valid and a **org.teiid.jdbc.AsynchPositioningException** will be thrown. That exception is recoverable if caught or can be avoided by calling **TeiidResultSet.available()** to determine if your desired positioning will be valid.

## 4.4. CONTINUOUS EXECUTION

The **RequestOptions** object may be used to specify a special type of continuous async execution via the **continuous** or **setContinuous** methods. In continuous mode the statement will be continuously re-executed. This is intended for consuming real-time or other data streams processed through a SQL plan. A continuous query will only terminate on an error or when the statement is explicitly closed. The SQL for a continuous query is no different than any other statement. Care should be taken to ensure that retrievals from non-continuous sources is appropriately cached for reuse, such as by using materialized views or session scoped temp tables. A continuous query must return a result set, must be executed with a forward-only resultset, and cannot be used in the scope of a transaction. Since resource consumptions is expected to be different in a continuous plan, it does not count against the server max active plan limit. Typically custom sources will be used to provide data streams.

When the client wishes to end the continuous query, the **Statement.close()** or **Statement.cancel()** method should be called. Typically your callback will close whenever it no long needs to process results.

See also the **ContinuousStatementCallback** for use as the **StatementCallback** for additional methods related to continuous processing.

## 4.5. EXECUTION PROPERTIES

The following table provides a list of execution properties as defined in **org.teiid.jdbc.ExecutionProperties**. These can be modified using the SET statement.

**Table 4.2. Execution Properties**

| Constant Identifier | String Value | Description |
|---|---|---|
| **ANSI_QUOTED_ IDENTIFIERS** | **ansiQuotedId entifiers** | See Section 2.8, "Connection Properties for the Driver and Data Source Classes". |
| **DISABLE_LOCA L_TRANSACTIO NS** | **disableLocal Txn** | See Section 2.8, "Connection Properties for the Driver and Data Source Classes". |

| Constant Identifier | String Value | Description |
| --- | --- | --- |
| `JDBC4COLUMNNAME ANDLABELSEMA NTICS` | `useJDBC4Colu mnName AndLabelSema ntics` | See Section 2.8, "Connection Properties for the Driver and Data Source Classes". |
| `NOEXEC` | See Section 2.8, "Connection Properties for the Driver and Data Source Classes". | |
| `PROP_FETCH_S IZE` | `fetchSize` | See Section 2.8, "Connection Properties for the Driver and Data Source Classes". |
| `PROP_PARTIAL _RESULTS_MOD E` | `partialResul tsMode` | See Section 2.8, "Connection Properties for the Driver and Data Source Classes" and Section 4.12, "Partial Results Mode". |
| `PROP_TXN_AUT O_WRAP` | `autoCommitTx n` | See Section 2.8, "Connection Properties for the Driver and Data Source Classes". |
| `PROP_XML_FOR MAT` | `XMLFormat` | Determines the formatting of XML documents returned by XML document models. Can be one of **XML_COMPACT_FORMAT** or **XML_TREE_FORMAT**. See Section 4.7, "XML Document Formatting" for more information. |
| `PROP_XML_VAL IDATION` | `XMLValidatio n` | Determines whether XML documents returned by XML document models will be validated against their schema after processing. See Section 4.8, "XML Schema Validation" and topics on "XML SELECT" in the *JBoss Data Virtualization Development Guide: Reference Material* for more information. |
| `QUERYTIMEOUT` | `QUERYTIMEOUT` | See Section 2.8, "Connection Properties for the Driver and Data Source Classes". |
| `RESULT_SET_C ACHE_MODE` | `resultSetCac heMode` | See Section 2.8, "Connection Properties for the Driver and Data Source Classes". |
| `SQL_OPTION_S HOWPLAN` | `SHOWPLAN` | See Section 2.8, "Connection Properties for the Driver and Data Source Classes". |

**See Also:**

- Section 4.9, "The SET Statement"

## 4.6. XML EXTENSIONS

JBoss Data Virtualization provides some XML extensions defined in
`org.teiid.jdbc.ExecutionProperties`:

- **PROP_XML_FORMAT** (defined as XMLFormat)

- **PROP_XML_VALIDATION** (defined as XMLValidation)

> **NOTE**
>
> These extensions apply to XML results from queries to XML document models, but not to XML produced by SQL/XML or read from some other source.

## 4.7. XML DOCUMENT FORMATTING

The **PROP_XML_FORMAT** execution property, defined in **org.teiid.jdbc.ExecutionProperties**, can be set using the SET statement to modify the way that XML documents are formatted from XML document models. The following valid values are also defined:

**XML_TREE_FORMAT**

XML_TREE_FORMAT (defined as Tree) returns a version of the XML formatted for display. The XML will use line breaks and tabs as appropriate to format the XML as a tree. This format is slower due to the formatting time and the larger document size.

**XML_COMPACT_FORMAT**

XML_COMPACT_FORMAT (defined as Compact) returns a version of the XML formatted for optimal performance. The XML is a single long string without any unnecessary white space.

> **IMPORTANT**
>
> If the **XML_COMPACT_FORMAT** execution property is not set, the formatting flag of the XML document in the original model is honored. This may produce either the tree or compact form of the document depending on the document setting.

## 4.8. XML SCHEMA VALIDATION

The **PROP_XML_VALIDATION** execution property, defined in **org.teiid.jdbc.ExecutionProperties**, can be set using the SET statement to indicate that the server should validate XML document model documents against their schema before returning them to the client. If schema validation is on, then the server sends an SQLWarning if the document does not conform to the schema it is associated with.

> **NOTE**
>
> Using schema validation will reduce the performance of your XML queries.

## 4.9. THE SET STATEMENT

Execution properties are set on the connection using the SET statement. The SET statement is not yet a language feature of JBoss Data Virtualization and is handled only in the JDBC client.

SET Syntax:

- SET [PAYLOAD] (parameter|SESSION AUTHORIZATION) value

- SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL (READ UNCOMMITTED|READ COMMITTED|REPEATABLE READ|SERIALIZABLE)

Syntax Rules:

- The parameter must be an identifier - it cannot contain spaces or other special characters only if quoted.

- The value may be either a non-quoted identifier or a quoted string literal value.

- If payload is specified, for example, **SET PAYLOAD x y**, then a session scoped payload properties object will have the corresponding name value pair set. The payload object is not fully session scoped. It will be removed from the session when the XAConnection handle is closed/returned to the pool (assumes the use of TeiidDataSource). The session scoped payload is superseded by usage of the **TeiidStatement.setPayload**.

Using SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is equivalent to calling Connection.setTransactionIsolation with the corresponding level.

The SET statement is most commonly used to control planning and execution.

- SET SHOWPLAN (ON|DEBUG|OFF)

- SET NOEXEC (ON|OFF)

The following is an example of how to use the SET statement to enable a debug plan:

```
Statement s = connection.createStatement();
s.execute("SET SHOWPLAN DEBUG");

Statement s1 = connection.createStatement();
ResultSet rs = s1.executeQuery("select col from table");

ResultSet planRs = s1.executeQuery("SHOW PLAN");
planRs.next();
String debugLog = planRs.getString("DEBUG_LOG");
Query Plan without executing the query
s.execute("SET NOEXEC ON");
s.execute("SET SHOWPLAN DEBUG");
...
e.execute("SET NOEXEC OFF");
```

The SET statement may also be used to control authorization. A SET SESSION AUTHORIZATION statement will perform a reauthentication (see Section 3.6, "Reauthentication") given the credentials currently set on the connection.

The connection credentials may be changed by issuing a SET PASSWORD statement.

```
Statement s = connection.createStatement();
s.execute("SET PASSWORD 'someval'");
s.execute("SET SESSION AUTHORIZATION 'newuser'");
```

## 4.10. THE SHOW STATEMENT

The SHOW statement can be used to see a variety of information. The SHOW statement is not yet a language feature of JBoss Data Virtualization and is handled only in the JDBC client.

**SHOW PLAN**

**SHOW PLAN** returns a resultset with a CLOB column PLAN_TEXT, an xml column PLAN_XML, and a CLOB column DEBUG_LOG with a row containing the values from the previously executed query. If SHOWPLAN is OFF or no plan is available, no rows are returned. If SHOWPLAN is not set to DEBUG, then DEBUG_LOG will return a null value.

**SHOW ANNOTATIONS**

**SHOW ANNOTATIONS** returns a resultset with string columns CATEGORY, PRIORITY, ANNOTATION, RESOLUTION and a row for each annotation on the previously executed query. If SHOWPLAN is OFF or no plan is available, no rows are returned.

**SHOW <property>**

**SHOW <property>** is the inverse of SET and shows the property value for the property supplied. It returns a resultset with a single string column with a name matching the property key.

**SHOW ALL**

**SHOW ALL** returns a resultset with a NAME string column and a VALUE string column with a row entry for every property value.

## 4.11. TRANSACTION STATEMENTS

In situations where direct use of the JDBC connection is not possible, transaction statements can be used to control a local transaction.

**START TRANSACTION**

START TRANSACTION is a synonym for **connection.setAutoCommit(false)**

**COMMIT**

COMMIT is a synonym for **connection.setAutoCommit(true)**

**ROLLBACK**

ROLLBACK is a synonym for **connection.rollback()** and returning to auto commit mode.

## 4.12. PARTIAL RESULTS MODE

JBoss Data Virtualization supports a partial results query mode. This mode changes the behavior of the query processor so the server returns results even when some data sources are unavailable.

For example, suppose that two data sources exist for different suppliers and your data designers have created a virtual group that creates a union between the information from the two suppliers. If your application submits a query without using partial results query mode and one of the suppliers' databases is down, the query against the virtual group returns an exception. However, if your application runs the same query in partial results query mode, the server returns data from the running data source and no data from the data source that is down.

When using partial results mode, if a source throws an exception during processing it does not cause the user's query to fail. Rather, that source is treated as returning no more rows after the failure point. Most commonly, that source will return 0 rows.

This behavior is most useful when using **UNION** or **OUTER  JOIN** queries as these operations handle missing information in a useful way. Most other kinds of queries will simply return 0 rows to the user when used in partial results mode and the source is unavailable.

> **NOTE**
>
> In some instances, (typically when you are using JDBC sources), if the source is not available initially, its absence will prevent Teiid from automatically determining the appropriate set of source capabilities. If you see an exception indicating that the capabilities for an unavailable source are not valid in partial results mode, then it may be necessary to manually set the database version or similar property on the translator to ensure that the capabilities are known even if the source is not available.

## 4.13. SETTING PARTIAL RESULTS MODE

Partial results mode is off by default but you can turn it on for all queries in a connection by using either setPartialResultsMode(true) on a DataSource or partialResultsMode=true on a JDBC URL. In either case, you can toggle partial results mode on or off later with a SET statement.

This is how you configure the partial results mode using the SET statement:

```
Statement statement = ...obtain statement from Connection...
statement.execute("set partialResultsMode true");
```

## 4.14. PARTIAL RESULTS WARNINGS

For each source that is excluded from the query, a warning will be generated describing the source and the failure. These warnings can be obtained from the **Statement.getWarnings()** method. This method returns a **SQLWarning** object, but in the case of partial results warnings this object will be an instance of the **org.teiid.jdbc.PartialResultsWarning** class. This class can be used to obtain a list of all the failed sources by name and to obtain the specific exception thrown by each resource adaptor.

> **NOTE**
>
> Since JBoss Data Virtualization supports cursoring before the entire result is formed, it is possible that a data source failure will not be determined until after the first batch of results have been returned to the client. This can happen in the case of unions, but not joins. To ensure that all warnings have been accumulated, the statement should be checked after the entire result set has been read.

The following is an example of how to obtain partial results warnings:

```
statement.execute("set partialResultsMode true");
ResultSet results = statement.executeQuery("SELECT Name FROM Accounts");
while (results.next())
{
    //process the result set
}
```

```
SQLWarning warning = statement.getWarnings();

if(warning instanceof PartialResultsWarning)
{
   PartialResultsWarning partialWarning = (PartialResultsWarning)warning;
   Collection failedConnectors = partialWarning.getFailedConnectors();
   Iterator iter = failedConnectors.iterator();
   while(iter.hasNext())
   {
      String connectorName = (String) iter.next();
      SQLException connectorException =
partialWarning.getConnectorException(connectorName);
      System.out.println(connectorName + ": " +
connectorException.getMessage());
   }
}
```

# CHAPTER 5. JDBC TRANSACTIONS

## 5.1. JDBC TRANSACTION TYPES

The JBoss Data Virtualization JDBC API supports three types of transactions from a client perspective:

- global transactions,

- local transactions, and

- request level transactions.

All are implemented by JBoss Data Virtualization as XA transactions. Refer to the JTA specification at http://www.oracle.com/technetwork/java/javaee/tech/jta-138684.html for more information on XA Transactions.

> **WARNING**
>
> The use of global, local, and request level transactions are all mutually exclusive. Request level transactions only apply when not in a global or local transaction. Any attempt to mix global and local transactions concurrently will result in an exception.

## 5.2. LOCAL TRANSACTIONS

A connection uses the **autoCommit** flag to explicitly control local transactions. By default, autoCommit is set to **true**, which indicates request level or implicit transaction control:

```
// Set auto commit to false and start a transaction
connection.setAutoCommit(false);

try
{
    // Execute multiple updates
    Statement statement = connection.createStatement();
    statement.executeUpdate("INSERT INTO Accounts (ID, Name) VALUES (10,
'Mike')");
    statement.executeUpdate("INSERT INTO Accounts (ID, Name) VALUES (15,
'John')");
    statement.close();

    // Commit the transaction
    connection.commit();
}
catch(SQLException e)
{
    // If an error occurs, rollback the transaction
    connection.rollback();
}
```

This example demonstrates several things:

1. Setting **autoCommit** flag to false. This will start a transaction bound to the connection.

2. Executing multiple updates within the context of the transaction.

3. When the statements are complete, the transaction is committed by calling **commit()**.

4. If an error occurs, the transaction is rolled back using the **rollback()** method.

## 5.3. ENDING LOCAL TRANSACTIONS

Any of the following operations will end a local transaction:

1. **Connection.setAutoCommit(true)** if previously set to false

2. **Connection.commit()**

3. **Connection.rollback()**

4. A transaction will be rolled back automatically if it times out.

## 5.4. TURNING OFF LOCAL TRANSACTIONS

In some cases, tools or frameworks above JBoss Data Virtualization will call **setAutoCommit(false)**, **commit()** and **rollback()** even when all access is read-only and no transactions are necessary. In the scope of a local transaction, JBoss Data Virtualization will start and attempt to commit an XA transaction, possibly complicating configuration or causing performance degradation.

In these cases, you can override the default JDBC behavior to indicate that these methods should perform no action regardless of the commands being executed. To turn off the use of local transactions, add the following property to the JDBC connection URL:

```
disableLocalTxn=true
```

> **WARNING**
>
> Turning off local transactions can be dangerous and can result in inconsistent results when reading or inconsistent data in data stores when writing. For safety, this mode should be used only if you are certain that the calling application does not need local transactions.

## 5.5. REQUEST LEVEL TRANSACTIONS

Request level transactions are used when the request is not in the scope of a global or local transaction, which implies **autoCommit** is **true**. In a request level transaction, your application does not need to explicitly call commit or rollback, rather every command is assumed to be its own transaction that will automatically be committed or rolled back by the server.

JBoss Data Virtualization can perform updates through virtual tables. These updates might result in an update against multiple physical systems, even though the application issues the update command against a single virtual table. Often, a user might not know whether the queried tables actually update multiple sources and require a transaction.

For that reason, JBoss Data Virtualization allows your application to automatically wrap commands in transactions when necessary. Because this wrapping incurs a performance penalty for your queries, you can choose from a number of available wrapping modes to suit your environment. You need to choose between the highest degree of integrity and performance your application needs. For example, if your data sources are not transaction-compliant, you might turn transaction wrapping off to maximize performance.

## 5.6. TRANSACTION WRAPPING MODES

You can set your transaction wrapping to one of the following modes:

1. **ON**

   This mode always wraps every command in a transaction without checking whether it is required. This is the safest mode.

2. **OFF**

   This mode never automatically wraps a command in a transaction or checks whether it needs to wrap a command. This mode can be dangerous as it will allow multiple source updates outside of a transaction without an error. This mode has best performance for applications that do not use updates or transactions.

3. **DETECT**

   This mode assumes that the user does not know how to execute multiple source updates in a transaction. JBoss Data Virtualization checks every command to see whether it is a multiple source update and wraps it in a transaction. If it is single source then it uses the source level command transaction.

## 5.7. SET THE TRANSACTION WRAPPING MODE

You can set the transaction mode as a property when you establish the connection using:

- the **autoCommitTxn** property in the connection URL (see Section 2.8, "Connection Properties for the Driver and Data Source Classes"),

- the **setAutoCommitTxn** method (see Section 2.8, "Connection Properties for the Driver and Data Source Classes"),

- or on a per-query basis, using the SET statement with the **PROP_TXN_AUTO_WRAP** property (see Section 4.5, "Execution Properties").

## 5.8. MULTIPLE INSERT BATCHES

If you work with request-level transactions and issue INSERTs with a query expression (or the deprecated SELECT INTO), you may find that multiple insert batches handled by separate source INSERTs are processed by the Red Hat JBoss Data Virtualization server. Take care to ensure that targeted sources support XA or that compensating actions are taken in the event of a failure.

## 5.9. GLOBAL TRANSACTIONS

Global or client XA transactions allow the JBoss Data Virtualization JDBC API to participate in transactions that are beyond the scope of a single client resource. For this, use the **org.teiid.jdbc.TeiidDataSource** class for establishing connections.

When the data source class is used in the context of a user transaction in an application server, such as JBoss, WebSphere, or Weblogic, the resulting connection will already be associated with the current XA transaction. No additional client JDBC code is necessary to interact with the XA transaction.

The following code demonstrates usage of UserTransactions.

```
UserTransaction ut = context.getUserTransaction();
try {
 ut.begin();
 Datasource oracle = lookup(...)
 Datasource teiid = lookup(...)
 Connection c1 = oracle.getConnection();
 Connection c2 = teiid.getConnection();
 // do something with Oracle connection
 // do something with Teiid connection
 c1.close();
 c2.close();
 ut.commit();
} catch (Exception ex) {
 ut.rollback();
}
```

The following code demonstrates manual usage of XA transactions.

```
XAConnection xaConn = null;
XAResource xaRes = null;
Connection conn = null;
Statement stmt = null;

try
{
   xaConn = <XADataSource instance>.getXAConnection();
   xaRes = xaConn.getXAResource();
   Xid xid = <new Xid instance>;
   conn = xaConn.getConnection();
   stmt = conn.createStatement();

   xaRes.start(xid, XAResource.TMNOFLAGS);
   stmt.executeUpdate("insert into â¦");
   // other statements on this connection or other resources enlisted in
this transaction
   // ...
   xaRes.end(xid, XAResource.TMSUCCESS);

   if (xaRes.prepare(xid) == XAResource.XA_OK)
   {
      xaRes.commit(xid, false);
   }
}
catch (XAException e)
```

```
{
    xaRes.rollback(xid);
}
finally
{
  // clean up code
  // ...
}
```

With the use of global transactions, multiple XAConnections may participate in the same transaction. It is important to note that the JDBC XAResource **isSameRM()** method only returns **true** if connections are made to the same server instance in a cluster. If the JBoss Data Virtualization connections are to different server instances then transactional behavior may not be the same as if they were to the same cluster member. For example, if the client transaction manager uses the same **XID** for each connection, duplicate XID exceptions may arise from the same physical source accessed through different cluster members. If the client transaction manager uses a different branch identifier for each connection, issues may arise with sources that lock or isolate changes based upon branch identifiers.

## 5.10. ENTERPRISE INFORMATION SYSTEM SUPPORT

The underlying resource adaptors that represent the Enterprise Information System (EIS) and the EIS itself must support XA transactions to be able to participate in distributed XA transaction using JBoss Data Virtualization. If a source system does not support the XA, then it can not participate in the distributed transaction. However, the source is still eligible to participate in data integration without XA support.

Participation in an XA transaction is automatically determined based on the resource adaptor's XA capability. It is the developer's responsibility to ensure they configure an XA resource when they require them to participate in distributed transactions.

# CHAPTER 6. CLIENT SSL CONNECTIONS

## 6.1. SSL CLIENT CONNECTIONS

You need to define certain properties for each SSL mode.
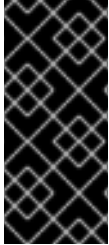
> **IMPORTANT**
>
> When connecting to the Red Hat JBoss Data Virtualization server with SSL enabled, you must use the "mms" protocol, instead of "mm" in the JDBC connection URL:
>
> ```
> jdbc:teiid:<myVdb>@mms://<host>:<port>
> ```

There are two different sets of properties that a client can configure to enable 1-way or 2-way SSL.

The first option is to use Java SSL properties. These are standard Java defined system properties to configure the SSL under any JVM, Teiid is not unique in its use of SSL. Provide the following system properties to the client VM process.

1-way SSL:

```
-Djavax.net.ssl.trustStore=<dir>/server.truststore (required)
-Djavax.net.ssl.trustStorePassword=<password> (optional)
-Djavax.net.ssl.keyStoreType (optional)
```

2-way SSL:

```
-Djavax.net.ssl.keyStore=<dir>/client.keystore (required)
-Djavax.net.ssl.keyStrorePassword=<password> (optional)
-Djavax.net.ssl.trustStore=<dir>/server.truststore (required)
-Djavax.net.ssl.trustStorePassword=<password> (optioanl)
-Djavax.net.ssl.keyStroreType=<keystore type> (optional)
```

The second option is to use Teiid-specific properties. Use this option when the above "javax" based properties are already in use by the host process. For example if your client application is a Tomcat process that is configured for https protocol and the above Java based properties are already in use, and importing Teiid-specific certificate keys into those https certificate keystores is not allowed.

In this scenario, a different set of Teiid-specific SSL properties can be set as system properties or defined inside the a "teiid-client-settings.properties" file. A sample "teiid-client-settings.properties" file can be found inside the "teiid-version-client.jar" file at the root called "teiid-client-settings.orig.properties". Extract this file, make a copy, change the property values required for the chosen SSL mode, and place this file in the client application's classpath before the "teiid-version-client.jar" file.

Here are the SSL properties and definitions that can be set in a **teiid-client-settings.properties** file:

```
###########################################
# SSL Settings
###########################################

#
# The key store type.  Defaults to JKS
```

```
#

org.teiid.ssl.keyStoreType=JKS

#
# The key store algorithm, defaults to
# the system property "ssl.TrustManagerFactory.algorithm"
#

#org.teiid.ssl.algorithm=

#
# The classpath or filesystem location of the
# key store.
#
# This property is required only if performing 2-way
# authentication that requires a specific private
# key.
#

#org.teiid.ssl.keyStore=

#
# The key store password (not required)
#

#org.teiid.ssl.keyStorePassword=

#
# The key alias(not required, if given named certificate is used)
#

#org.teiid.ssl.keyAlias=

#
# The key password(not required, used if the key password is different
than the keystore password)
#

#org.teiid.ssl.keyPassword=

#
# The classpath or filesystem location of the
# trust store.
#
# This property is required if performing 1-way
# authentication that requires trust not provided
# by the system defaults.
#

#org.teiid.ssl.trustStore=

#
# The trust store password (not required)
#
```

```
#org.teiid.ssl.trustStorePassword=

#
# The cipher protocol, defaults to TLSv3
#

org.teiid.ssl.protocol=TLSv1

#
# Whether to allow anonymous SSL
# (the TLS_DH_anon_WITH_AES_128_CBC_SHA cipher suite)
# defaults to true
#

org.teiid.ssl.allowAnon=true
1-way SSL
org.teiid.ssl.trustStore=<dir>/server.truststore (required)
2-way SSL
org.teiid.ssl.keyStore=<dir>/client.keystore (required)
org.teiid.ssl.trustStore=<dir>/server.truststore (required)
```

# CHAPTER 7. USING HIBERNATE WITH JBOSS DATA VIRTUALIZATION

## 7.1. CONFIGURE HIBERNATE FOR USE WITH JBOSS DATA VIRTUALIZATION

**Prerequisites**

- You must have the JBoss Data Virtualization JDBC API client JAR file (**teiid-client.jar**) and the JBoss Data Virtualization hibernate dialect JAR file (**teiid-hibernate-dialect-*VERSION*.jar**) in Hibernate's classpath. These files are found in ***EAP_HOME*/modules/system/layers/dv/org/jboss/teiid/client/main/**.

  These are required for the **org.teiid.dialect.TeiidDialect**, **org.teiid.jdbc.TeiidDriver** and **org.teiid.jdbc.TeiidDataSource** classes.

**Procedure 7.1. Configure Hibernate for Use with JBoss Data Virtualization**

1. **Open the Hibernate configuration file**
   Open the **hibernate.cfg.xml** file.

2. **Specify the JBoss Data Virtualization driver class**
   Specify the JBoss Data Virtualization driver class in the **connection.driver_class** property:

   ```
   <property name="connection.driver_class">
      org.teiid.jdbc.TeiidDriver
   </property>
   ```

3. **Set the Connection URL**
   Specify the URL for the VDB in the **connection.url** property:

   ```
   <property name="connection.url">
      jdbc:teiid:VDB-
   NAME@mm://HOST:POST;user=USERNAME;password=PASSWORD
   </property>
   ```

   > **NOTE**
   >
   > Be sure to use a local connection if Hibernate is in the same VM as the application server. .

4. **Specify the dialect class**
   Specify the JBoss Data Virtualization dialect class in the **dialect** property:

   ```
   <property name="dialect">
      org.teiid.dialect.TeiidDialect
   </property>
   ```

**NOTE**

Alternatively, the connection properties can be added to the **hibernate.properties** file instead of **hibernate.cfg.xml**:

```
hibernate.connection.driver_class=org.teiid.jdbc.TeiidDriver
hibernate.connection.url=jdbc:teiid:VDB-NAME@mm://HOST:PORT
hibernate.connection.username=USERNAME
hibernate.connection.password=PASSWORD
hibernate.dialect=org.teiid.dialect.TeiidDialect
```

**NOTE**

Since your VDBs will likely contain multiple source and view models with identical table names, you will need to fully qualify table names specified in Hibernate mapping files:

```
<class name="CLASSNAME" table="SOURCE/VIEW_MODEL_NAME.
[SCHEMA_NAME.]TABLENAME">
    ...
</class>
```

For example:

```
<class name="org.teiid.example.Publisher"
table="BOOKS.BOOKS.PUBLISHERS">
    ...
</class>
```

## 7.2. LIMITATIONS OF USING HIBERNATE WITH JBOSS DATA VIRTUALIZATION

Many Hibernate use cases assume a data source has the ability (with proper user permissions) to process Data Definition Language (DDL) statements like CREATE TABLE and DROP TABLE as well as Data Manipulation Language (DML) statements like SELECT, UPDATE, INSERT and DELETE. JBoss Data Virtualization can handle a broad range of DML, but does not directly support DDL against a particular source.

Sequence and Identity generation are not supported. Identifier generation based upon table values, such as the hilo generator, require that the identifier table(s) be exposed through JBoss Data Virtualization. The GUID identifier generation strategy is directly supported.

# CHAPTER 8. ODATA SUPPORT

## 8.1. WHAT IS ODATA?

The Open Data Protocol (OData) is a Web protocol for querying and updating data that provides a way to unlock your data and free it from silos that exist in applications today. OData does this by applying and building upon Web technologies such as HTTP, Atom Publishing Protocol (AtomPub) and JSON to provide access to information from a variety of applications, services, and stores. OData is used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems and traditional Web sites.

OData is consistent with the way the Web works - it makes a deep commitment to URIs for resource identification and commits to an HTTP-based, uniform interface for interacting with those resources (just like the Web). This allows OData to enable a new level of data integration and interoperability across a broad range of clients, servers, services, and tools.

## 8.2. SUPPORT FOR ODATA

When a user successfully deploys a VDB into a JBoss Data Virtualization server, the OData protocol support is implicitly provided by the JBoss Data Virtualization server without any further configuration. OData support is currently not available in the JBoss Data Virtualization Embedded profile. JBoss Data Virtualization makes use of JBoss EAP and its already configured RestEasy and OData4J libraries to provide Rest based access to the VDB. The access would be similar to accessing to any web resources deployed on JBoss EAP. OData support is implemented and deployed through a single WAR file across the JBoss Data Virtualization system. A user can access this WAR file at `EAP_HOME/dataVirtualization/vdb/teiid-odata-VERSION.war`.

## 8.3. HOW TO ACCESS THE DATA?

This section describes how to access the data. For example, assume you have a vdb by name northwind deployed and that vdb has a table customers in a model called NW. You can access the table as:

```
http://localhost:8080/odata/northwind.1/NW.customers
```

This is similar to making a JDBC/ODBC connection and issuing a SQL call as

```
SELECT * FROM NW.customers
```

**NOTE**

You need to fully qualify the table name along with the model name. Also, use correct case (upper or lower) as used in the VDB.

The returned results from OData query can be in Atom/AtomPub XML format or JSON format. By default AtomPub based XML result is returned.

> **NOTE**
>
> When you issue the above query if you receive a message similar to:
>
> ```
> <error
> xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices/me
> tadata">
> <code>NotFoundException</code>
> <message lang="en-US">EdmEntitySet NW.customer is not
> found</message>
> </error>
> ```
>
> Then, it means that either you supplied the model-name.table-name combination wrong, in that case, check spelling and case. Or your table did not have any PRIMARY KEY or UNIQUE KEY(s) on them. Since OData access is more key oriented, it is mandatory that every table Teiid exposes through OData interface must have a PK or at least one UNIQUE key.
>
> Teiid does support composite PRIMARY KEY too.

If you are not able to see all the rows, then see the configuration section below for more details. Issue another call as:

```
http://localhost:8080/odata/northwind.1/NW.customers?$skiptoken=256
```

You can also submit criteria with along the query to filter the results:

```
http://localhost:8080/odata/northwind.1/NW.customers?$filter=name eq 'bob'
```

This is similar to making a JDBC/ODBC connection and issuing a SQL call as:

```
SELECT * FROM NW.customers where name = 'bob'
```

The power of OData querying comes from navigations from one entity to another, similar to the foreign key relationships in the relational databases. For example, if customers table had a association with orders table, where say customers table had primary key of ID then, you can issue a query:

```
http://localhost:8080/odata/northwind.1/NW.customers(1234)/NW.orders?
$filter=orderdate > '12-31-2012'
```

This is similar to making a JDBC/ODBC connection and issuing a SQL call as:

```
SELECT * FROM NW.orders o join NW.customers c join o.customer_id = c.id
where c.id=1234 and o.orderdate > '12-31-2012'
```

## 8.4. SECURITY

Currently by default OData access is secured using HTTPBasic authentication. The user will be authenticated against JBoss Data Virtualization's default security domain teiid-security and makes use of the security role odata. However, if you wish to change the security domain, manually edit the **web.xml** WAR file in ***EAP_HOME*/dataVirtualization/vdb/teiid-odata-*VERSION*.war** directory.

## 8.5. CONFIGURATION

The OData WAR file can be configured with following properties in the `web.xml` file.

| Property Name | Description | Default Value |
| --- | --- | --- |
| batch-size | Number of rows to send back each time, -1 returns all rows | 256 |
| skiptoken-cache-time | Time interval between the results being recycled/expired between $skiptoken requests | 300000 |
| local-transport-name | JBoss Data Virtualization's Local transport name for connection | odata |

JBoss Data Virtualization OData server, implements cursoring logic when the result rows exceed the configured batch size. On every request, only batch-size number of rows are returned. Each such request is considered an active cursor, with a specified amount of idle time specified by skip-token-cache-time. After the cursor is timed out, the cursor will be closed and remaining results will be cleaned up, and will no longer be available for further queries. Since there is no session based tracking of these cursors, if the request for skiptoken comes after the expired time, the original query will be executed again and tries to reposition the cursor to relative absolute potion, however the results are not guaranteed to be same as the underlying sources may have been updated with new information meanwhile.

## 8.6. LIMITATIONS

The following feature limitations currently apply.

- JBoss Data Virtualization implements the OData V2 specification along with many V3 features, however OData V3 is not completely supported.

- Blob support for media types are not supported.

- $value construct to retrieve individual column value is not supported.

- create/update/delete $links is not supported.

- $expand is not supported.

## 8.7. CLIENT TOOLS FOR ACCESS

OData access is really where the user comes in, depending upon your programming model and needs there are various ways you write your access layer into OData. The following are some suggestions:

- Your Browser: The OData Explorer is an online tool for browsing an OData data service.

- Microsoft .NET Framework 3.51: the WCF Data Services framework is available as a separate download for .NET 3.x.

- Microsoft .NET Framework 4.0: the WCF Data Services framework built into .NET 4.0.

- Silverlight 3: the Data Services client library for Silverlight is available for download.

- Java: the Restlet 2.0 library for Java (including Java on your Android phone) supports the OData protocol.

- Java: Use a library like OData4J for Java based access, or any Rest based framework

- JavaScript: the XMLHttpRequest object is standard in modern browsers or you can use jQuery, which comes out of the box with .NET 4.0 or is available for download.

- PHP: the Toolkit for PHPprovides OData support for PHP clients.

- AJAX: if you're using AJAX for ASP.NET, Microsoft provides the ASP.NET Ajax Library for getting to OData.

- Excel 2010 PowerPivot: PowerPivot comes with OData support built right in.

- Windows Desktop: LINQPad is a good tool for building OData queries interactively but it has some limitations: firstly, it is not able to handle FunctionImports (procedures). If the model contains only procedures, no tables then also it acts error. Secondly, it does not work with multiple schema, as it does not show all the schemas, only the default one. Since no default schema is set in the VDB's $metadata, it finds the first one and uses it. OData V2 does allow multiple Schemas and multiple EntityContainers in a single $metadata so it appears that LINQPad is not yet fully compliant with this specification.

- Shell Scripts: use CURL tool

## 8.8. HOW JBOSS DATA VIRTUALIZATION EXPOSES SCHEMA FOR ODATA?

OData defines its schema using Conceptual Schema Definition Language (CSDL). Every VDB, that is deployed in an ACTIVE state in JBoss Data Virtualization server exposes its metadata in CSDL format. For example if you want retrieve metadata for your vdb northwind, you need to issue a query like

```
http://localhost:8080/odata/northwind/$metadata
```

Since OData schema model is not a relational schema model, JBoss Data Virtualization uses the following semantics to map its relational schema model to OData schema model.

| Relational Entity | Mapped OData Entity |
| --- | --- |
| Model Name | Schema Namespace, EntityContainer Name |
| Table/View | EntityType, EntitySet |
| Table Columns | EntityType's Properties |
| Primary Key | EntityType's Key Properties |
| Foreign Key | Navigation Property on EntityType, Association, AssosiationSet |

| Relational Entity | Mapped OData Entity |
| --- | --- |
| Procedure | FunctionImport |
| Procedure's Table Return | ComplexType |

JBoss Data Virtualization by design does not define any embedded ComplexType in the EnitityType.

> **WARNING**
>
> JBoss Data Virtualization does not define any one EntityContainer that resulted from different vdb models as a default container, so all entities must be accessed using full path to them.

# APPENDIX A. UNSUPPORTED JDBC METHODS

## A.1. UNSUPPORTED JDBC METHODS

This appendix lists those JDBC methods that JBoss Data Virtualization does not support, based on the JDK 1.7 JDBC. Unless specified below, JBoss Data Virtualization supports all other JDBC Methods.

Those methods listed without comments throw an exception of type **SQLException** stating that it is not supported.

Where specified, some listed methods do not throw an exception, but possibly exhibit unexpected behavior. If no arguments are specified, then all related (overridden) methods are not supported. If an argument is listed then only those forms of the method specified are not supported.

## A.2. RESULTSET LIMITATIONS

- **TYPE_SCROLL_SENSITIVE** is not supported.

- **UPDATABLE ResultSet**s are not supported.

- Returning multiple **ResultSet**s from Procedure execution is not supported.

## A.3. UNSUPPORTED CLASSES AND METHODS IN JAVA.SQL

**Table A.1. Connection Properties**

| Class name | Methods |
| --- | --- |
| **Array** | `Not Supported` |
| **Blob** | `getBinaryStream(long, long) - throws SQLFeatureNotSupportedException`<br>`setBinaryStream(long) - - throws SQLFeatureNotSupportedException`<br>`setBytes - - throws SQLFeatureNotSupportedException`<br>`truncate(long) - throws SQLFeatureNotSupportedException` |

| Class name | Methods |
|---|---|
| CallableStatement | getArray - throws SQLFeatureNotSupportedException<br>getObject(int parameterIndex, Map&lt;String, Class&lt;?&gt;&gt; map) - throws SQLFeatureNotSupportedException<br>getRef - throws SQLFeatureNotSupportedException<br>getRowId - throws SQLFeatureNotSupportedException<br>getURL(String parameterName) - throws SQLFeatureNotSupportedException<br>registerOutParameter - ignores<br>registerOutParameter(String parameterName, *) - throws SQLFeatureNotSupportedException<br>setRowId(String parameterName, RowId x) - throws SQLFeatureNotSupportedException<br>setURL(String parameterName, URL val) - throws SQLFeatureNotSupportedException |
| Clob | getCharacterStream(long arg0, long arg1) - throws SQLFeatureNotSupportedException<br>setAsciiStream(long arg0) - throws SQLFeatureNotSupportedException<br>setCharacterStream(long arg0) - throws SQLFeatureNotSupportedException<br>setString - throws SQLFeatureNotSupportedException<br>truncate - throws SQLFeatureNotSupportedException |
| Connection | createArrayOf - throws SQLFeatureNotSupportedException<br>createBlob - throws SQLFeatureNotSupportedException<br>createClob - throws SQLFeatureNotSupportedException<br>createNClob - throws SQLFeatureNotSupportedException<br>createSQLXML - throws SQLFeatureNotSupportedException<br>createStruct(String typeName, Object[] attributes) - throws SQLFeatureNotSupportedException<br>getClientInfo - throws SQLFeatureNotSupportedException<br>releaseSavepoint - throws SQLFeatureNotSupportedException<br>rollback(Savepoint savepoint) - throws SQLFeatureNotSupportedException<br>setHoldability - throws SQLFeatureNotSupportedException<br>setSavepoint - throws SQLFeatureNotSupportedException<br>setTypeMap - throws SQLFeatureNotSupportedException |

| Class name | Methods |
|---|---|
| `DatabaseMeta Data` | `getAttributes - throws SQLFeatureNotSupportedException`<br>`getClientInfoProperties  - throws SQLFeatureNotSupportedException`<br>`getFunctionColumns - throws SQLFeatureNotSupportedException`<br>`getFunctions - throws SQLFeatureNotSupportedException`<br>`getRowIdLifetime - throws SQLFeatureNotSupportedException` |
| `NClob` | `Not Supported` |
| `PreparedStat ement` | `setArray - throws SQLFeatureNotSupportedException`<br>`setRef - throws SQLFeatureNotSupportedException`<br>`setRowId - throws SQLFeatureNotSupportedException`<br>`setUnicodeStream - throws SQLFeatureNotSupportedException` |
| `Ref` | `Not Implemented` |
| `ResultSet` | `deleteRow - throws SQLFeatureNotSupportedException`<br>`getArray - throws SQLFeatureNotSupportedException`<br>`getHoldability - throws SQLFeatureNotSupportedException`<br>`getObject(*, Map&lt;String, Class&lt;? &gt;&gt; map) - throws SQLFeatureNotSupportedException`<br>`getRef - throws SQLFeatureNotSupportedException`<br>`getRowId - throws SQLFeatureNotSupportedException`<br>`getUnicodeStream - throws SQLFeatureNotSupportedException`<br>`getURL - throws SQLFeatureNotSupportedException`<br>`insertRow - throws SQLFeatureNotSupportedException`<br>`moveToInsertRow - throws SQLFeatureNotSupportedException`<br>`refreshRow - throws SQLFeatureNotSupportedException`<br>`rowDeleted - throws SQLFeatureNotSupportedException`<br>`rowInserted - throws SQLFeatureNotSupportedException`<br>`rowUpdated - throws SQLFeatureNotSupportedException`<br>`setFetchDirection - throws SQLFeatureNotSupportedException`<br>`update*  - throws SQLFeatureNotSupportedException` |
| `RowId` | `Not Supported` |
| `Savepoint` | `not Supported` |
| `SQLData` | `Not Supported` |

| Class name | Methods |
|---|---|
| **SQLInput** | `not Supported` |
| **SQLOutput** | `Not Supported` |
| **Statement** | `setCursorName(String)` |
| **Struct** | `Not Supported` |

## A.4. UNSUPPORTED CLASSES AND METHODS IN JAVAX.SQL

**Table A.2. Connection Properties**

| Class name | Methods |
|---|---|
| **RowSet*** | `Not Supported` |

# APPENDIX B. KEYTOOL

## B.1. KEYTOOL

Keytool is an encryption key and certificate management utility. It enables users to create and manage their own public/private key pairs and associated certificates for use in self-authentication, and also to cache public keys (in the form of certificates) belonging to other parties, for securing communication to those parties.

## B.2. USING KEYTOOL WITH JBOSS DATA VIRTUALIZATION

When using the keytool to manage public key cryptography for JBoss Data Virtualization, use the following options:

- Set the alias to **teiid** using the **-alias teiid** option.

- Set the algorithm to **RSA** using the **-keyslg RSA** option.

- Set the validity period to **365** days using the **-validity 365** option.

- Set the store type to **JKS** using the **-storetype JKS** option.

## B.3. CREATE A PRIVATE/PUBLIC KEY PAIR WITH KEYTOOL

**Procedure B.1. Create a Private/Public Key Pair with Keytool**

1. Run the **keytool -genkey -alias *ALIAS* -keyalg *ALGORITHM* -validity *DAYS* - keystore *server.keystore* -storetype *TYPE*** command:

   ```
   keytool -genkey -alias teiid -keyalg RSA -validity 365 -keystore
   server.keystore -storetype JKS
   ```

2. If the specified keystore already exists, enter the existing password for that keystore, otherwise enter a new password:

   ```
   Enter keystore password: <password>
   ```

3. Answer the following questions when prompted:

   ```
   What is your first and last name?
   [Unknown]:  <userâÂÂs name>
   What is the name of your organizational unit?
   [Unknown]:  <department name>
   What is the name of your organization?
   [Unknown]:  <company name>
   What is the name of your City or Locality?
   [Unknown]:  <city name>
   What is the name of your State or Province?
   [Unknown]:  <state name>
   What is the two-letter country code for this unit?
   [Unknown]:  <country name>
   ```

4. Enter **yes** to confirm the provided information is correct:

```
Is CN=<userÃ¢ÂÂs name>, OU=<department name>, O="<company name>",
L=<city name>, ST=<state name>, C=<country name>  correct?
[no]:  yes
```

5. Enter your desired keystore password:

```
Enter key password for <server>
(Return if same as keystore password)
```

**Result**

The `server.keystore` file contains the newly generated public and private key pair.

# B.4. EXTRACT A SELF-SIGNED CERTIFICATE FROM THE KEYSTORE

**Procedure B.2. Extract a Self-signed Certificate from the Keystore**

1. Run the **keytool -export -alias *ALIAS* -keystore *server.keystore* -rfc -file *public.cert*** command:

```
keytool -export -alias teiid -keystore server.keystore -rfc -file
public.cert
```

2. Enter the keystore password when prompted:

```
Enter keystore password: <password>
```

**Result**

This creates the `public.cert` file that contains a certificate signed with the private key in the `server.keystore`.

# B.5. ADD A CERTIFICATE TO A TRUSTSTORE USING KEYTOOL

**Procedure B.3. Add a Certificate to a Truststore Using Keytool**

1. Run the **keytool -import -alias *ALIAS* -file *public.cert* -storetype *TYPE* -keystore *server.truststore*** command:

```
keytool -import -alias teiid -file public.cert -storetype JKS -
keystore server.truststore
```

2. If the specified truststore already exists, enter the existing password for that truststore, otherwise enter a new password:

```
Enter keystore password:  <password>
```

3. Enter **yes** when prompted to trust the certificate:

```
Owner: CN=<user's name>, OU=<dept name>, O=<company name>, L=<city>,
ST=<state>, C=<country>
Issuer: CN=<user's name>, OU=<dept name>, O=<company name>, L=
<city>, ST=<state>, C=<country>
Serial number: 416d8636
Valid from: Fri Jul 31 14:47:02 CDT 2009 until: Sat Jul 31 14:47:02
CDT 2010
Certificate fingerprints:
        MD5:  22:4C:A4:9D:2E:C8:CA:E8:81:5D:81:35:A1:84:78:2F
        SHA1:
05:FE:43:CC:EA:39:DC:1C:1E:40:26:45:B7:12:1C:B9:22:1E:64:63
Trust this certificate? [no]:  yes
```

**Result**

The certificate in **public.cert** has been added to the new truststore named **server.truststore**.

# APPENDIX C. REVISION HISTORY

**Revision 6.2.0-57**        **Thu Dec 10 2015**        **David Le Sage**

Updates for 6.2.