Teiid - Scalable Information Integration

1

Teiid Client Developer's Guide

7.0

Connecting to Teiid Server

The Teiid JDBC API provides Java Database Connectivity (JDBC) access to any Virtual Database (VDB) deployed on a Teiid Server. The Teiid JDBC API is compatible with the JDBC 4.0 specification; however, it does not fully support all *methods*. Advanced features, such as updatable result sets or SQL3 data types, are not supported.

Java client applications connecting to a Teiid Server will need to use Java 1.6 JDK. Previous versions of Java are not supported.

Before you can connect to the Teiid Server using the Teiid JDBC API, please do following tasks first.

1. Install the Teiid Server. See the "Admin Guide" for instructions.
2. Build a Virtual Database (VDB). You can either build a "Dynamic VDB" (Designer not required), or you can use the Eclipse based GUI tool *Designer* [http://www.jboss.org/teiiddesigner.html]. Check the "Reference Guide" for instructions on how to build a VDB. If you do not know what VDB is, then start with this *document* [http://www.jboss.org/teiid/basics/virtualdatabases.html].
3. Deploy the VDB into Teiid Server. Check "Admin Guide" for instructions.
4. Start the Teiid Server (JBoss AS), if it is not already running.

Now that you have the VDB deployed in Teiid Server, client applications can connect to Teiid Server and issue SQL queries against deployed VDB using Teiid's JDBC API. If you are new to JDBC, learn about *JDBC* [http://java.sun.com/docs/books/tutorial/jdbc/index.html] here. Teiid ships with `teiid-7.0-client.jar` in the "`jboss-install/server/<profile>/lib`" directory.

Main classes in the client JAR:

- `org.teiid.jdbc.TeiidDriver` - allows JDBC connections using the *DriverManager* [http://java.sun.com/javase/6/docs/api/java/sql/DriverManager.html] class.
- `org.teiid.jdbc.TeiidDatasource` - allows JDBC connections using the *DataSource* [http://java.sun.com/javase/6/docs/api/javax/sql/DataSource.html] or *XADataSource* [http://java.sun.com/javase/6/docs/api/javax/sql/XADataSource.html] class. You should use this class to create managed or XA connections.

Once you have established a connection with the Teiid Server, you can use standard JDBC API classes, like DatabaseMetadata and ResultSetMetadata, to interrogate metadata and Statement classes to execute queries.

1.1. Driver Connection

Use `org.teiid.jdbc.TeiidDriver` as the driver class.

Use the following URL format for JDBC connections:

```
jdbc:teiid:<vdb-name>@mm[s]://<host>:<port>;[prop-name=prop-value;]*
```

URL Components

1. <vdb-name> - Name of the VDB you are connecting to
2. mm - defines Teiid JDBC protocol, mms defines a secure channel (see the *SSL chapter* for more)
3. <host> - defines the server where the Teiid Server is installed
4. <port> - defines the port on which the Teiid Server is listening for incoming JDBC connections.
5. [prop-name=prop-value] - additionally you can supply any number of name value pairs separated by semi-colon [;]. All supported URL properties are defined in the *connection properties section*. Property values should be URL encoded if they contain reserved characters, e.g. ('?', '=', ';', etc.)

1.1.1. URL Connection Properties

The following table shows all the supported connection properties that can used with Teiid JDBC Driver URL connection string, or on the Teiid JDBC Data Source class.

Table 1.1. Connection Properties

Property Name	Type	Description
`ApplicationName`	`String`	Name of the client application; allows the administrator to identify the connections
`FetchSize`	`int`	Size of the resultset; The default size if 500. <=0 indicates that the default should be used.
`partialResultsMode`	`boolean`	Enable/disable support partial results mode. Default false. See the *partial results* section.
`autoCommitTxn`	`String`	Only applies only when "autoCommit" is set to "true". This determines how a executed command needs to be transactionally wrapped inside the Teiid engine to maintain the data integrity. • ON - Always wrap command in distributed transaction • OFF - Never wrap command in distributed transaction • DETECT (default)- If the executed command is spanning more than one source it automatically uses distributed transaction.

Property Name	Type	Description
		Transactions with JDBC for more information.
`disableLocalTxn`	`boolean`	If "true", the autoCommit setting, commit and rollback will be ignored for local transactions. Default false.
`user`	`String`	User name
`Password`	`String`	Credential for user
`ansiQuotedIdentifiers`	`boolean`	Sets the parsing behavior for double quoted entries in SQL. The default, true, parses dobuled quoted entries as identifiers. If set to false, then double quoted values that are valid string literals will be parsed as string literals.
`version`	`integer`	Version number of the VDB
`resultSetCacheMode`	`boolean`	ResultSet caching is turned on/off. Default false.
`autoFailover`	`boolean`	If true, will automatically select a new server instance after a communication exception. Default false. This is typically not needed when connections are managed, as the connection can be purged from the pool.
`SHOWPLAN`	`String`	(typically not set as a connection property) Can be ON|OFF|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 OFF.
`NoExec`	`String`	(typically not set as a connection property) Can be ON|OFF; ON prevents query execution, but parsing and planning will still occur. Default OFF.

1.2. Datasource Connection

To use a data source based connection, use `org.teiid.jdbc.TeiidDataSource` as the data source class. The `TeiidDataSource` is also an XADatasource. Teiid DataSource class is also Serializable, so it possible for it to be used with JNDI naming services.

Note

Teiid supports the XA protocol, XA transactions will be extended to Teiid sources that also support XA.

All the properties (except for version, which is known on TeiidDataSource as DatabaseVersion) defined in the *connection properties*have corresponding "set" methods on the `org.teiid.jdbc.TeiidDataSource`. Properties that are assumed from the URL string have addtional "set" methods, which are described in the following table.

Table 1.2. Datasource Properties

Property Name	Type	Description
`DatabaseName`	`String`	The name of a virtual database (VDB) deployed to Teiid
`ServerName`	`String`	Server where the Teiid runtime installed
`AdditionalProperties`	`String`	Optional setting of properties that has the same format as the property string in a connection URL.
`PortNumber`	`integer`	Port number on which the Server process is listening on.
`secure`	`boolean`	Secure connection. Flag to indicate to use SSL (mms) based connection between client and server
`DatabaseVersion`	`integer`	VDB version
`DataSourceName`	`String`	Name given to this data source

1.3. Standalone Application

To use either Driver or DataSource based connections, add the client JAR to your Java client application's classpath. See the simple client example in the kit for a full Java sample of the following.

1.3.1. Driver Connection

Sample Code:

```
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);
   }
}
```

1.3.2. Datasource Connection

Sample Code:

```
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();
    }
}
```

1.4. JBoss AS Data Source

Teiid can be configured as a JDBC data source in the JBoss Application Server to be accessed from JNDI or injected into your JEE applications. Deploying Teiid as data source in JBoss AS is exactly same as deploying any other RDBMS resources like Oracle or DB2.

Defining as data source is not limited to JBoss AS, you can also deploy as data source in Glassfish, Tomcat, Websphere, Weblogic etc servers, however their configuration files are different than JBoss AS. Consult the respective documentation of the environment in which you are deploying.

Installation Steps

1. If Teiid is not installed in the AS instance, copy the `teiid-7.0-client.jar` into `<jboss-install>/server/<profile>/lib` directory.
2. Create a "<datasource name>-ds.xml" file in `<jboss-install>/server/<profile>/deploy` directory. Based on the type of deployment (XA, driver, or local), the contents of the file will be different. See the following sections for more.

Tje data source will then be accessable through the JNDI name specified in the -ds.xml file.

1.4.1. DataSource Connection

Make sure you know the correct DatabaseName, ServerName, Port number and credentials that are specific to your deployment environment.

Example 1.1. Sample XADataSource in the JBoss AS using the Teiid DataSource class `org.teiid.jdbc.TeiidDataSource`

```
<datasources>
  <xa-datasource>
    <jndi-name>TEIID-DS</jndi-name>
    <xa-datasource-class>org.teiid.jdbc.TeiidDataSource</xa-datasource-class>
    <xa-datasource-property name="DatabaseName">myVDB</xa-datasource-property>
```

```
    <xa-datasource-property name="serverName">localhost</xa-datasource-property>
    <xa-datasource-property name="portNumber">31000</xa-datasource-property>
    <xa-datasource-property name="user">admin</xa-datasource-property>
    <xa-datasource-property name="password">password</xa-datasource-property>
    <track-connection-by-tx>true</track-connection-by-tx>

    <isSameRM-override-value>false</isSameRM-override-value>
    <no-tx-separate-pools />

    <!-- pool and other JBoss datasource properties -->
    <check-valid-connection-sql>SELECT 1</check-valid-connection-sql>
    <min-pool-size>5</min-pool-size>
    <max-pool-size>10</max-pool-size>
  </xa-datasource>
</datasources>
```

1.4.2. Driver based connection

You can also use Teiid's JDBC driver class `org.teiid.jdbc.TeiidDriver` to create a data source

```
<datasources>
  <local-tx-datasource>
    <jndi-name>TEIID-DS</jndi-name>
    <connection-url>jdbc:metamatrix:myVDB@mm://localhost:31000</connection-url>
    <driver-class>org.teiid.jdbc.TeiidDriver</driver-class>
    <user-name>admin</user-name>
    <password>teiid</password>

    <!-- pool and other JBoss datasource properties -->
    <check-valid-connection-sql>SELECT 1</check-valid-connection-sql>
    <min-pool-size>5</min-pool-size>
    <max-pool-size>10</max-pool-size>
  </local-tx-datasource>
</datasources>
```

1.4.3. Local JDBC Connection

If you are deploying your client application on the same JBoss AS instance as the Teiid runtime is installed, then there is no reason for your client application to open socket based JDBC connection. You can use slightly modified data source configuration to make a "local" connection, where the JDBC API will lookup a local Teiid runtime in the same VM.

Example 1.2. Local data source

```
<datasources>
  <xa-datasource>
    <jndi-name>TEIID-DS</jndi-name>
    <xa-datasource-class>org.teiid.jdbc.TeiidDataSource</xa-datasource-class>
    <xa-datasource-property name="DatabaseName">myVDB</xa-datasource-property>
    <xa-datasource-property name="user">admin</xa-datasource-property>
    <xa-datasource-property name="password">password</xa-datasource-property>
    <track-connection-by-tx>true</track-connection-by-tx>
    <isSameRM-override-value>false</isSameRM-override-value>
    <no-tx-separate-pools />

    <!-- pool and other JBoss datasource properties -->
    <check-valid-connection-sql>SELECT 1</check-valid-connection-sql>
    <min-pool-size>5</min-pool-size>
    <max-pool-size>10</max-pool-size>
  </xa-datasource>
</datasources>
```

This is essentially the same as the XA configuration, but "ServerName" and "PortNumber" are not specified.

Teiid extensions to the JDBC API

2.1. Statement Extensions

The Teiid statement extension interface, `org.teiid.jdbc.TeiidStatement`, provides functionality beyond the JDBC standard. To use the extension interface, simply cast or unwap the statement returned by the Connection. The following methods are provided on the extension interface:

Table 2.1. Connection Properties

Method Name	Description
`getAnnotations`	Get the query engine annotations if the statement was last executed with SHOWPLAN ON|DEBUG. Each `org.teiid.client.plan.Annotation` contains a description, a category, a severity, and possibly a resolution of notes recorded during query planning that can be used to understand choices made by the query planner.
`getDebugLog`	Get the debug log if the statement was last executed with SHOWPLAN DEBUG.
`getExecutionProperty`	Get the current value of an execution property on this statement object.
`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`	Set the execution property on this statement. See the *execution properties* section for more information. It is generally preferable to use the *SET statement* unless the execution property applies only to the statement being executed.
`setPayload`	Set a per-command payload to pass to translators. Currently the only built-in use is for sending hints for Oracle data source.

2.2. Execution Properties

Execution properties may be set on a per statement basis through the *`TeiidStatement`* interface or on the connection via the *SET statement*. For convenience, the property keys are defined by constants on the `org.teiid.jdbc.ExecutionProperties` interface.

Table 2.2. Execution Properties

Property Name/String Constant	Description
`PROP_TXN_AUTO_WRAP / autoCommitTxn`	Same as the connection property.
`PROP_PARTIAL_RESULTS_MODE / partialResultsMode`	See the *partial results* section.
`PROP_XML_FORMAT / XMLFormat`	Determines the formatting of XML documents returned by XML document models. See the *document formatting* section.
`PROP_XML_VALIDATION / XMLValidation`	Determines whether XML documents returned by XML document models will be validated against their schema after processing. See the *document validation* section.
`RESULT_SET_CACHE_MODE / resultSetCacheMode`	Same as the connection property.
`SQL_OPTION_SHOWPLAN / SHOWPLAN`	Same as the connection property.
`NOEXEC / NOEXEC`	Same as the connection property.

2.3. Set Statement

Execution properties may also be set on the connection by using the SET statement. The SET statement is not yet a language feature of Teiid and is handled only in the JDBC client.

SET Syntax:

- SET parameter value

Syntax Rules:

- Both parameter and value must be simple literals - they cannot contain spaces.
- The value is also not treated as an expression and will not be evaluated prior to being set as the parameter value.

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

- SET SHOWPLAN (ON|DEBUG|OFF)

- SET NOEXEC (ON|OFF)

Example 2.1. Enabling Plan Debug

```
Statement s = connection.createStatement();
s.execute("SET SHOWPLAN DEBUG");
...
Statement s1 = connection.createStatement();
ResultSet rs = s1.executeQuery("select col from table");
TeiidStatement ts = s1.unwrap(TeiidStatement.class);
String debugLog = ts.getDebugLog();
```

2.4. Partial Results Mode

The Teiid Server 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.

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 `ResultSet.getWarnings()` method. This method returns a `SQLWarning` object but in the case of "partial results" warnings, this will be an object of type `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.

Below is an example of printing the list of failed sources:

```
statement.setExecutionProperty(ExecutionProperties.PROP_PARTIAL_RESULTS_MODE,
 “true”);
```

```
ResultSet results = statement.executeQuery(“SELECT Name FROM Accounts”);
SQLWarning warning = results.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();
    }
}
```

2.5. XML extensions

The XML extensions apply on to XML resutls from queries to XML document models, and not to XML produced by SQL/XML or read from some other source.

2.5.1. Document formatting

The PROP_XML_FORMAT execution property can be set to modify the way that XML documents are formatted from XML document models. Valid values for the constant are defined in the same ExecutionProperties interface:

1. `XML_TREE_FORMAT` - 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.
2. `XML_COMPACT_FORMAT` - Returns a version of the XML formatted for optimal performance. The XML is a single long string without any unnecessary white space.
3. Not Set - If no format is set, the formatting flag on 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.

2.5.2. Schema validation

The `PROP_XML_VALIDATION` execution property can be set 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 send a SQLWarning if the document does not conform to the schema it is associated with. Using schema validation will reduce the performance of your XML queries.

Transactions with JDBC

The Teiid JDBC API supports three types of transactions from a client perspective – global, local, and request level. All are implemented by the Teiid Server as XA transactions. See the *JTA specification* [http://java.sun.com/javaee/technologies/jta/index.jsp] for more on XA Transactions.

3.1. Local Transactions

The Connection class uses the "autoCommit" flag to explicitly control local transactions. By default, autoCommit is set to "true", which indicates request level or implicit transaction control. example of how to use local transactions by setting the autoCommit flag to false.

Example 3.1. Local transaction control using autoCommit

```
// 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.

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.

3.1.1. Turning Off Local Transactions

In some cases, tools or frameworks above Teiid 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 Teiid 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 this property to the JDBC connection URL

```
disableLocalTxn=true
```

Warning

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

3.2. 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.

The Teiid Server 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, the Teiid Server 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 the transaction wrapping off (completely) to maximize performance.

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 check 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 to execute multiple source updates in a transaction. The Teiid Server checks every command to see whether it is a multiple source update and wraps it in a transaction. If it is single source then uses the source level command transaction.

You can set the transaction mode as a property when you establish the Connection or on a per-query basis using the execution properties. For more information on execution properties, see the section *"Execution Properties"*

3.2.1. Multiple Insert Batches

When issuing an INSERT with a query expression (or the deprecated SELECT INTO), multiple insert batches handled by separate source INSERTS may be processed by the Teiid server. Care should be taken to ensure that targeted sources support XA or that compensating actions are taken in the event of a failure.

3.3. Using Global Transactions

Global or client XA transactions allow the Teiid JDBC API to participate in transactions that are beyond the scope of a single client resource. For this use the Teiid DataSource Class for establishing connections.

When the DataSource is used in the context of a UserTransaction 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.

Example 3.2. >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 …");
  <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>
}
```

With the use of global transactions multiple Teiid XAConnections may participate in the same transaction. It is important to note that the Teiid JDBC XAResource "isSameRM" method only returns "true", if connections are made to the same server instance in a cluster. If the Teiid 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.

3.4. Restrictions

3.4.1. Application Restrictions

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.

3.4.2. Enterprise Information System Support

The underlying resource adaptors that represent the EIS system and the EIS system itself must support XA transactions if they want to participate in distributed XA transaction thru Teiid. If source

system do not support the XA, then it can not particilate in the distributed transaction. However, the source is still eligible to participate in data integration with out the XA support

The participation in the XA transaction is automatically determined based on the resource adaptors XA capability. It is user's repsonsiblity to make sure that they configure a XA resource when they require them to participate in distributed transaction.

SSL Client Connections

This chapter will shows you various security configurations that can be used with Teiid in securing your data access. Note that data level security (*data roles*) are explained in separate chapter.

4.1. Default Security

If you are always using a *local connection*, then you do need to secure a channels.

By default all sensitive (non-data) messages between client and server are encrypted using a *Diffy-Hellman* [http://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange] key that is negotiated per connection. This encryption is controlled by `clientEncryptionEnabled` property in `JdbcSslConfiguration` and `AdminSslConfiguration` sections in `<jboss-install>/server/<profile>/deploy/teiid/teiid-jboss-beans.xml` file.

4.2. SSL Modes

Teiid supports SSL based channel between the client JDBC application and Teiid Server. Teiid supports the following SSL modes.

1. Anonymous – No certificates are required, but all communications are still encrypted using the TLS_DH_anon_WITH_AES_128_CBC_SHA SSL suite.
2. 1-way – Only authenticates the server to the client traffic. Requires a private key keystore to be created for the server and a truststore at the client that authenticates that key. The SSL suite is negotiated.
3. 2-way – Mutual client and server authentication. The server and client applications each have a keystore for their private keys and each has a truststore that authenticates the other.

Depending upon the SSL mode, follow the guidelines of your organization around creating/obtaining private keys. If you have no organizational requirements, then follow this guide to create *self-signed certificates* with their respective keystores and truststores.

The following keystore and truststore combinations are required for different SSL modes. The names of the files can be chosen by the user. The following files are shown for example purposes only.

1-way

1. server.keystore - has server's private key
2. server.truststore - has server's public key

2-way

1. server.keystore - has server's private key
2. server.truststore - has server's public key
3. client.keystore - client's private key
4. client.truststore - has client's public key

4.3. Client SSL Settings

The following sections define the properties required for each SSL mode. Note that when connecting to Teiid Server with SSL enabled, you *MUST* use the *"mms"* protocol, instead of "mm" in the JDBC connection URL, for example

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.

4.3.1. Option 1: 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.

Example 4.1. 1-way SSL

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

Example 4.2. 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)
```

4.3.2. Option 2: Teiid Specific Properties

Use this option for *anonymous* mode or 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 "teiid-client-settings.properties" file. The "teiid-client-settings.properties" file can be found inside the "teiid-7.0-client.jar" file at the root. Extract this file, or 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-7.0-client.jar" file.

SSL properties and definitions inside the "teiid-client-settings.properties" are shown below.

```
########################################
# 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 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.
#
# Set to NONE for anonymous authentication using
# the TLS_DH_anon_WITH_AES_128_CBC_SHA cipher suite
#

#org.teiid.ssl.trustStore=

#
# The trust store password (not required)
#

#org.teiid.ssl.trustStorePassword=

#
# The cipher protocol, defaults to SSLv3
#

org.teiid.ssl.protocol=SSLv3
```

Example 4.3. 1-way SSL

```
org.teiid.ssl.trustStore=<dir>/server.truststore (required)
```

Example 4.4. 2-way SSL

```
org.teiid.ssl.keyStore=<dir>/client.keystore (required)
org.teiid.ssl.trustStore=<dir>/server.truststore (required)
```

Example 4.5. Anonymous

```
org.teiid.ssl.trustStore=NONE
```

Using Teiid with Hibernate

5.1. Limitations

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. Teiid can handle a broad range of DML, but does not support DDL.

5.2. Configuration

For the most part, interacting with Teiid VDBs (Virtual Databases) from Hibernate is no different from working with any other type of data source. First you must place Teiid JDBC API client JAR file and Teiid's hibernate dialect JAR in Hibernate's classpath. These files can be found in `<jboss-install>/server/<profile>/lib` directory.

- teiid-7.0-client.jar
- teiid-hibernate-dialect-7.0.jar

These JAR files have the `org.teiid.dialect.TeiidDialect` and `org.teiid.jdbc.TeiidDriver` and `org.teiid.jdbc.TeiidDataSource` classes.

You then configure Hibernate (via hibernate.cfg.xml) as follows:

1. Specify the Teiid driver class in the "connection.driver_class" property:

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

2. Specify the URL for the VDB in the "connection.url" property (replacing terms in angle brackets with the appropriate values):

```
<property name="connection.url">
                                jdbc:metamatrix:<vdb-name>@mm://<host>:<port>;user=<user-
name>;password=<password>
</property>
```

3. Specify the Teiid dialect class in the “dialect” property:

```
<property name="dialect">
```

```
 org.teiid.dialect.TeiidDialect
</property>
```

Alternatively, if you put your connection properties in `hibernate.properties` instead of `hibernate.cfg.xml`, they would look like this:

```
hibernate.connection.driver_class=org.teiid.jdbc.TeiidDriver
hibernate.connection.url=jdbc:teiid:<vdb-name>@mm://<host>:<port>
hibernate.connection.username=<user-name>
hibernate.connection.password=<password>
hibernate.dialect=org.teiid.dialect.TeiidDialect
```

Note also that 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="<Class name>" table="<Source/view model name>.[<schema name>.]<Table
 name>">
 ...
</class>
```

Example 5.1. Example Mapping

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

Appendix A. Unsupported JDBC Methods

Based upon the JDBC in JDK 1.6, this appendix details only those JDBC methods that Teiid does not support. Unless specified below, Teiid supports all other JDBC Methods.

Those methods listed without comments throw a 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.1. ResultSet Limitations

- TYPE_SCROLL_SENSITIVE is not supported.
- `UPDATABLE` ResultSets are not supported.
- Returning multiple ResultSets from Procedure execution is not supported.

A.2. Unsupported Classes and Methods in "java.sql"

Table A.1. Connection Properties

Class name	Methods
`Array`	`Not Supported`
`Blob`	getBinaryStream(long, long) - throws SQLFeatureNotSupportedException setBinaryStream(long) - - throws SQLFeatureNotSupportedException setBytes - - throws SQLFeatureNotSupportedException truncate(long) - throws SQLFeatureNotSupportedException
`CallableStatement`	getArray - throws SQLFeatureNotSupportedException getBigDecimal(String parameterName)- throws SQLFeatureNotSupportedException getBlob(String parameterName)- throws SQLFeatureNotSupportedException

Class name	Methods
	getBoolean(String parameterName)- throws SQLFeatureNotSupportedException getByte(String parameterName)- throws SQLFeatureNotSupportedException getBytes(String parameterName)- throws SQLFeatureNotSupportedException getCharacterStream(String parameterName)- throws SQLFeatureNotSupportedException getClob(String parameterName)- throws SQLFeatureNotSupportedException getDate(String parameterName, *)- throws SQLFeatureNotSupportedException getDouble(String parameterName)- throws SQLFeatureNotSupportedException getFloat(String parameterName)- throws SQLFeatureNotSupportedException getInt(String parameterName)- throws SQLFeatureNotSupportedException getLong(String parameterName)- throws SQLFeatureNotSupportedException getNCharacterStream - throws SQLFeatureNotSupportedException getNClob - throws SQLFeatureNotSupportedException getNString - throws SQLFeatureNotSupportedException getObject(int parameterIndex, Map<String, Class<?>> map) - throws SQLFeatureNotSupportedException getObject(String parameterName) - throws SQLFeatureNotSupportedException getRef - throws SQLFeatureNotSupportedException getRowId - throws SQLFeatureNotSupportedException getShort(String parameterName) - throws SQLFeatureNotSupportedException getSQLXML(String parameterName) - throws SQLFeatureNotSupportedException getString(String parameterName) - throws SQLFeatureNotSupportedException getTime(String parameterName, *) - throws SQLFeatureNotSupportedException getTimestamp(String parameterName, *) - throws SQLFeatureNotSupportedException getURL(String parameterName) - throws SQLFeatureNotSupportedException

Class name	Methods
	registerOutParameter - ignores registerOutParameter(String parameterName, *) - throws SQLFeatureNotSupportedException setAsciiStream - throws SQLFeatureNotSupportedException setBigDecimal(String parameterName, BigDecimal x)- throws SQLFeatureNotSupportedException setBinaryStream(String parameterName, *) - throws SQLFeatureNotSupportedException setBlob(String parameterName, *)- throws SQLFeatureNotSupportedException setBoolean(String parameterName, boolean x) - throws SQLFeatureNotSupportedException setByte(String parameterName, byte x) - throws SQLFeatureNotSupportedException setBytes(String parameterName, byte[] x) - throws SQLFeatureNotSupportedException setCharacterStream - throws SQLFeatureNotSupportedException setClob(String parameterName, *) - throws SQLFeatureNotSupportedException setDate(String parameterName, *) - throws SQLFeatureNotSupportedException setDouble(String parameterName, double x) - throws SQLFeatureNotSupportedException setFloat(String parameterName, float x) - throws SQLFeatureNotSupportedException setLong(String parameterName, long x) - throws SQLFeatureNotSupportedException setNCharacterStream - throws SQLFeatureNotSupportedException setNClob - throws SQLFeatureNotSupportedException setNString - throws SQLFeatureNotSupportedException setNull - throws SQLFeatureNotSupportedException setObject(String parameterName, *) - throws SQLFeatureNotSupportedException setRowId(String parameterName, RowId x) - throws SQLFeatureNotSupportedException setSQLXML(String parameterName, SQLXML xmlObject) - throws SQLFeatureNotSupportedException setShort(String parameterName, short x) - throws SQLFeatureNotSupportedException setString(String parameterName, String x) - throws SQLFeatureNotSupportedException

Class name	Methods
	setTime(String parameterName, *) - throws SQLFeatureNotSupportedException setTimestamp(String parameterName, *) - throws SQLFeatureNotSupportedException setURL(String parameterName, URL val) - throws SQLFeatureNotSupportedException
`Clob`	getCharacterStream(long arg0, long arg1) - throws SQLFeatureNotSupportedException setAsciiStream(long arg0) - throws SQLFeatureNotSupportedException setCharacterStream(long arg0) - throws SQLFeatureNotSupportedException setString - throws SQLFeatureNotSupportedException truncate - throws SQLFeatureNotSupportedException
`Connection`	createArrayOf - throws SQLFeatureNotSupportedException createBlob - throws SQLFeatureNotSupportedException createClob - throws SQLFeatureNotSupportedException createNClob - throws SQLFeatureNotSupportedException createSQLXML - throws SQLFeatureNotSupportedException createStatement(int resultSetType,int resultSetConcurrency, int resultSetHoldability) - throws SQLFeatureNotSupportedException createStruct(String typeName, Object[] attributes) - throws SQLFeatureNotSupportedException getClientInfo - throws SQLFeatureNotSupportedException prepareCall(String sql, int resultSetType,int resultSetConcurrency, int resultSetHoldability) - throws SQLFeatureNotSupportedException prepareStatement(String sql, int autoGeneratedKeys) - throws SQLFeatureNotSupportedException prepareStatement(String sql, int[] columnIndexes) - throws SQLFeatureNotSupportedException prepareStatement(String sql, String[] columnNames) - throws SQLFeatureNotSupportedException

Class name	Methods
	releaseSavepoint - throws SQLFeatureNotSupportedException rollback(Savepoint savepoint) - throws SQLFeatureNotSupportedException setHoldability - throws SQLFeatureNotSupportedException setSavepoint - throws SQLFeatureNotSupportedException setTypeMap - throws SQLFeatureNotSupportedException
`DatabaseMetaData`	getAttributes - throws SQLFeatureNotSupportedException getClientInfoProperties - throws SQLFeatureNotSupportedException getFunctionColumns - throws SQLFeatureNotSupportedException getFunctions - throws SQLFeatureNotSupportedException getRowIdLifetime - throws SQLFeatureNotSupportedException
`NClob`	`Not Supported`
`PreparedStatement`	execute(String sql) - throws SQLException executeQuery(String sql) - throws SQLException executeUpdate(String sql) - throws SQLException setArray - throws SQLFeatureNotSupportedException setNCharacterStream - throws SQLFeatureNotSupportedException setNClob - throws SQLFeatureNotSupportedException setRef - throws SQLFeatureNotSupportedException setRowId - throws SQLFeatureNotSupportedException setUnicodeStream - throws SQLFeatureNotSupportedException
`Ref`	`Not Implemented`
`ResultSet`	deleteRow - throws SQLFeatureNotSupportedException getArray - throws SQLFeatureNotSupportedException

Class name	Methods
	getAsciiStream - throws SQLFeatureNotSupportedException getHoldability - throws SQLFeatureNotSupportedException getNCharacterStream - throws SQLFeatureNotSupportedException getNClob - throws SQLFeatureNotSupportedException getNString - throws SQLFeatureNotSupportedException getObject(*, Map<String, Class<?>> map) - throws SQLFeatureNotSupportedException getRef - throws SQLFeatureNotSupportedException getRowId - throws SQLFeatureNotSupportedException getUnicodeStream - throws SQLFeatureNotSupportedException getURL - throws SQLFeatureNotSupportedException insertRow - throws SQLFeatureNotSupportedException moveToInsertRow - throws SQLFeatureNotSupportedException refreshRow - throws SQLFeatureNotSupportedException rowDeleted - throws SQLFeatureNotSupportedException rowInserted - throws SQLFeatureNotSupportedException rowUpdated - throws SQLFeatureNotSupportedException setFetchDirection - throws SQLFeatureNotSupportedException update* - throws SQLFeatureNotSupportedException
RowId	Not Supported
Savepoint	not Supported
SQLData	Not Supported
SQLInput	not Supported
SQLOutput	Not Supported
Statement	execute(String, int) execute(String, int[]) execute(String, String[]) executeUpdate(String, int) executeUpdate(String, int[]) executeUpdate(String, String[]) getGeneratedKeys() getResultSetHoldability() setCursorName(String)

Class name	Methods
Struct	Not Supported

A.3. Unsupported Classes and Methods in "javax.sql"

Table A.2. Connection Properties

Class name	Methods
RowSet*	Not Supported
StatementEventListener	Not Supported

Appendix B. Generating Self Signed Certificate with Keytool

To generate a self-signed certificate, you need a program called “keytool”, which is supplied with any version of the Java SDK. The instructions below walk through the creation of both the key store and the trust store files for a 1-way SSL configuration with the security keys.

B.1. Creating private/public key pair:

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

 Enter keystore password:  <enter password>
 What is your first and last name?
 [Unknown]:  <user’s 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>

 Is CN=<user’s name>, OU=<department name>, O="<company name>",
 L=<city name>, ST=<state name>, C=<country name>  correct?
 [no]:  yes
 Enter key password for <server>
 (Return if same as keystore password)
```

The "server.keystore" can be used as keystore based upon the newly created private key.

B.2. Extracting the public key

From the "server.keystore" created above we can extract a public key for creating a trust store

```
keytool -export -alias teiid –keystore server.keystore -rfc -file public.cert
 Enter keystore password: <enter passsword>
```

This creates the "public.cert" file that contains the public key based on the private key in the "server.keystore"

B.3. Creating the Truststore

```
keytool -import -alias teiid -file public.cert –storetype JKS -keystore server.truststore
Enter keystore password:  <enter password>
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
```

Now this has created "server.truststore". There are many other ways to create self signed certificates, the above procedure is just one way. If you would like create them using "openssl", see *this tutorial* [http://www.akadia.com/services/ssh_test_certificate.html].