Teiid - Scalable Information Integration

1

Teiid Connector Developer's Guide

6.2.0

Connectors in Teiid

In Teiid a connector handles all communications with individual enterprise information sources, which can include databases, data feeds, flat files, or any other entity you have modeled.

In Teiid, a connector is used to:

- Translate a Teiid-specific command into a native command.
- Execute the command.
- Return batches of results to Teiid.

Teiid is responsible for reassembling the results from one or more connectors into an answer for the user's command.

For a more detailed workflow, see the chapter *"Connector API."*

1.1. Do You Need a New Connector?

Teiid can provide several connectors for common enterprise information system types. If you can use one of these enterprise information systems, you do not need to develop a custom one.

Teiid offers the following connectors:

- *JDBC:* Connects to many relational databases. The JDBC Connector is validated against the following database systems: Oracle, Microsoft SQL Server, IBM DB2, MySQL, Postgres, Derby, and Sybase. In addition, the JDBC Connector can often be used with other 3rd-party drivers and provides a wide range of extensibility options to specialize behavior against those drivers.
- *Text:* Connects to text files.
- *XML* Connects to XML files on disk or by invoking Web services on other enterprise systems.
- *LDAP* Connects to LDAP directory services.
- *Salesforce* Connects to Salesforce.

1.2. Required Items to Write a Custom Connector

To write a connector, follow this procedure:

1. Gather all necessary information about your Enterprise Information System (EIS). You will need to know:
   - API for accessing the system

- Configuration and connection information for the system
- Expectation for incoming queries/metadata
- The SQL and processing constructs supported by information system.
- Required properties for the connector, such as URL, user name, etc.
- The CDK development kit (jars and tools).

2. Implement the required interfaces defined by the Connector API.
   - Connector – starting point.
   - Connection – represents a connection to the source.
   - ConnectorCapabilities – specifies what kinds of commands your connector can execute
   - Execution (and sub-interfaces) – specifies how to execute each type of command
3. Test your connector with Connector Development Kit (CDK) test utilities.
4. Deploy your connector type into Teiid.
   - Create and import your connector type definition file.
   - Create a connector binding using the connector type
   - Deploy a Virtual Database with metadata corresponding to your EIS
5. Execute queries via Teiid.

This guide covers how to do each of these steps in detail. It also provides additional information for advanced topics, such as connection pooling, streaming large objects, and transactions. For a sample connector code, please check the *Teiid community* [http://teiid.org]

Connector API

2.1. Overview

A component called the Connector Manager is controlling access to your connector. This chapter reviews the basics of how the Connector Manager interacts with your connector while leaving reference details and advanced topics to be covered in later chapters.

A custom connector must implement the following interfaces to connect and query an enterprise Data Source. These interfaces are in package called *org.teiid.connector.api:*

- *Connector* - This interface is the starting point for all interaction with your connector. It allows the Connector Manager to obtain a connection and perform lifecycle events.
- *Connection* - This interface represents a connection to your data source. It is used as a starting point for actual command executions. Connections provided to the Connector Manager will be obtained and released for each command execution. Teiid provides for extensible automatic connection pooling, as discussed in the *Connection Pooling* chapter.
- *ConnectorCapabilities* - This interface allows a connector to describe the execution capabilities of the connector.
- *Execution (and sub-interfaces)* - These interfaces represent a command execution with your Connector. There is a sub-interface for executing each kinds of command: ResultSetExecution, UpdateExecution, and ProcedureExecution.

Note that many of the interfaces above have base implementations in the *org.teiid.connector.basic* package. Consider extending the corresponding BasicXXX class rather than fully implementing the interface.

The most important interfaces provided by Teiid to the connector are the following:

- *ConnectorEnvironment* – an interface describing access to external resources for your connector.
- *ConnectorLogger* – an interface for writing logging information to Teiid logs.
- *ExecutionContext* – interface defining the execution context available to the connector when executing a command.

2.2. Connector Lifecycle

2.2.1. Starting

A Connector instance will be initialized one time via the start method, which passes in a *ConnectorEnvironment* object provided by the *Connector Manager* . The *ConnectorEnvironment provides* the following resources to the connector:

- Configuration properties – name / value pairs as provided by the connector binding in the Teiid Console
- Logging – ConnectorLogger interface allows a Connector to log messages and errors to Teiid's log files.
- Type facility – an interface defining runtime datatypes and type conversion facility.
- Scheduling facility – repeating tasks can be scheduled and managed by Teiid.
- Caching facility – easy methods for caching based upon relevant contexts, such as session or query scope.

2.2.2. Running

While the connector is running it is expected to return provide connections and capabilities information in response to system requests. If the source system is not available ConnectorExceptions or RuntimeExceptions may be thrown at any time to indicate failure. The connector should handle failure internally in a graceful manner, since the system will not automatically perform a stop/start.

2.2.3. Stopping

The stop method will be called on system shutdown or on an administrative call that to stop the connector. Once a connector has been stopped the instance is removed from the system. A new Connector instance will be created prior to the start call.

2.3. Connections to Source

2.3.1. Obtaining connections

The connector must implement the getConnection() method to allow the Connector Manager to obtain a connection. The getConnection() method is passed a ExecutionContext, which contains information about the context in which this query is being executed.

The ExecutionContext provides the following:

- User name
- Virtual database name
- Virtual database version
- The ability to add execution warnings.
- Trusted token

The trusted token is used to pass security information specific to your application through the Teiid. The client can pass the trusted token when they connect via JDBC. This token is then passed to

the Membership Service and may be created, replaced, or modified at that time. In some cases, you may wish to provide a customer Membership Service implementation to handle security needs specific to your organization. For more information on implementing see the *Server Extension Guide* [http://www.jboss.org/teiid//docs.html]

2.3.2. Releasing Connections

Once the Connector Manager has obtained a connection, it will use that connection only for the lifetime of the request. When the request has completed, the close() method will be called on the connection.

In cases (such as when a connection is stateful and expensive to create), connections should be pooled. Teiid provides an extensible connection pool for this purpose, as described in chapter *Connection Pooling*.

2.4. Executing Commands

2.4.1. Execution Modes

The Connector API uses a Connection to obtain an execution interface for the command it is executing. The actual queries themselves are sent to connectors in the form of a set of objects, which are further described in Chapter *Command Language*. Connectors are allowed to support any subset of the available execution modes.

Table 2.1. Types of Execution Modes

Execution Interface	Command interface(s)	Description
`ResultSetExecution`	`IQueryCommand`	A query corresponding to a SQL SELECT or set query statement.
`UpdateExecution`	`IInsert`, `IUpdate`, `IDelete`, `IBatchedUpdates`	An insert, update, or delete, corresponding to a SQL INSERT, UPDATE, or DELETE command
`ProcedureExecution`	`IProcedure`	A procedure execution that may return a result set and/or output values.

All of the execution interfaces extend the base `Execution` interface that defines how executions are cancelled and closed. ProcedureExecution also extends ResultSetExecution, since procedures may also return resultsets.

2.4.2. ResultSetExecution

Typically most commands executed against connectors are IQueryCommands. While the command is being executed, the connector provides results via the ResultSetExecution next method. The next method should return null to indicate the end of results. Note: the expected

batch size can be obtained from the ExecutionContext and used as a hint in fetching results from the EIS.

2.4.3. Update Execution

Each execution returns the update count(s) expected by the update command. If possible IBatchedUpdates should be executed atomically. The ExecutionContext can be used to determine if the execution is already under a transaction.

2.4.4. Procedure Execution

Procedure commands correspond to the execution of a stored procedure or some other functional construct. A procedure takes zero or more input values and can return a result set and zero or more output values. Examples of procedure execution would be a stored procedure in a relational database or a call to a web service.

If a result set is expected when a procedure is executed, all rows from it will be retrieved via the ResultSetExecution interface first. Then, if any output values are expected, they will be retrieved via the getOutputParameterValues() method.

2.4.5. Asynchronous Executions

In some scenarios, a connector needs to execute asynchronously and allow the executing thread to perform other work. To allow this, you should:

- Set either the SynchronousWorkers annotation or the connector binding property SynchWorkers to false - this overrides the default behavior in which connector threads stay associated with their Execution until the Execution is closed.
- Throw a DataNotAvailableExecption during a retrival method, rather than explicitly waiting or sleeping for the results. The DataNotAvailableException may take a delay parameter in its constructor to indicate how long the system should wait befor polling for results. Any non-negative value is allowed.
- Be aware that a connector with asynchronous workers cannot be transactional.

2.4.6. Bulk Execution

Non batched `IInsert`, `IUpdate`, `IDelete` commands may have Iliteral values marked as multiValued if the ConnectorCapabilities shows support for BulkUpdate. Commands with multiValued Iliterals represent multiple executions of the same command with different values. As with IBatchedUpdates, bulk operations should be executed atomically if possible.

2.4.7. Command Completion

All normal command executions end with the calling of close() on the Execution object. Your implementation of this method should do the appropriate clean-up work for all state in the Execution object.

2.4.8. Command Cancellation

Commands submitted to Teiid may be aborted in several scenarios:

- Client cancellation via the JDBC API (or other client APIs)
- Administrative cancellation
- Clean-up during session termination
- Clean-up if a query fails during processing

Unlike the other execution methods, which are handled in a single-threaded manner, calls to cancel happen asynchronously with respect to the execution thread.

Your connector implementation may choose to do nothing in response to this cancellation message. In this instance, Teiid will call close() on the execution object after current processing has completed. Implementing the cancel() method allows for faster termination of queries being processed and may allow the underlying data source to terminate its operations faster as well.

2.5. Monitored Connectors

Teiid can automatically monitor connectors, which will update a status flag on the connector. This status can be checked via the AdminApi and is exposed in the console. To use connector monitoring effectively:

- Set a positive test interval value on on the connector binding (default is 600) indicating the number of seconds between status checks. These checks are useful in idle periods.
- Implement a meaningful isAlive method on your Connector Connections.
- Use either a pooled Connector or a Connector that supports single identity.

Possible status results include:

- Not initialized - to indicate not yet started.
- Init failed - to indicate start failed.
- Open - to indicate the running state and that connections can be obtained form the source.
- Unable to check - to indicate the running state but connections cannot be obtained administratively.
- Data Source Unavailable - to indicate the running state.
- Closed - to indicate that the Connector has been stopped.

Command Language

3.1. Language Interfaces

Teiid sends commands to your connector in object form. The interfaces for these objects are all defined in the org.teiid.connector.language package. These interfaces can be combined to represent any possible command that Teiid may send to the connector. However, it is possible to notify Teiid that your connector can only accept certain kinds of commands via the ConnectorCapabilities class. See the section on using *Connector Capabilities* for more information.

The language interfaces all extend from the ILanguageObject interface. Language objects should be thought of as a tree where each node is a language object that has zero or more child language objects of types that are dependent on the current node.

All commands sent to your connector are in the form of these language trees, where the root of the tree is a subclass of ICommand. ICommand has several sub-interfaces, namely: IQueryCommand, IInsert, IUpdate, IDelete, IBatchedUpdate, and IProcedure. Important components of these commands are expressions, criteria, and joins, which are examined in closer detail below. Also see the *Teiid JavaDocs* [http://docs.jboss.org/teiid/6.2.0/apidocs] for more on the classes and interfaces described here.

3.1.1. Expressions

An expression represents a single value in context, although in some cases that value may change as the query is evaluated. For example, a literal value, such as 5 represents an integer value. An element reference such as "table.EmployeeName" represents a column in a data source and may logically take on many values while the command is being evaluated.

- `IExpression` – base expression interface
- `IElement` – represents an element in the data source
- `ILiteral` – represents a literal scalar value, but may also be multi-valued in the case of bulk updates.
- `IFunction` – represents a scalar function with parameters that are also IExpressions
- `IAggregate` – represents an aggregate function which holds a single expression
- `IScalarSubquery` – represents a subquery that returns a single value
- `ISearchedCaseExpression` – represents a searched CASE expression. The searched CASE expression evaluates the criteria in WHEN clauses till one evaluates to TRUE, then evaluates the associated THEN clause.

3.1.2. Criteria

A criteria is a combination of expressions and operators that evaluates to true, false, or unknown. Criteria are most commonly used in the WHERE or HAVING clauses.

- `ICriteria` – the base criteria interface
- `ILogicalCriteria` – used to logically combine other criteria
- `INotCriteria` – used to NOT another criteria
- `ICompoundCriteria` – used to combine other criteria via AND or OR
- `IPredicateCriteria` – a predicate that evaluates to true, false, or unknown
- `ISubuqeryCompareCriteria` – represents a comparison criteria with a subquery including a quantifier such as SOME or ALL
- `ICompareCriteria` – represents a comparison criteria with =, >, <, etc.
- `IBaseInCriteria` – base class for an IN criteria
- `IInCriteria` – represents an IN criteria that has a set of expressions for values
- `ISubqueryInCriteria` – represents an IN criteria that uses a subquery to produce the value set
- `IIsNullCriteria` – represents an IS NULL criteria
- `IExistsCriteria` – represents an EXISTS criteria that determines whether a subquery will return any values
- `ILikeCriteria` – represents a LIKE criteria that compares string values

3.1.3. The FROM Clause

The FROM clause contains a list of IFromItems. Each IFomItem can either represent a group or a join between two other IFromItems. This allows joins to be composed into a join tree.

- `IGroup` – represents a single group
- `IJoin` – has a left and right `IFromItem` and information on the join between the items
- `IInlineView` – represents a group defined by an inline `IQueryCommand`

A list of IFromItems is used by default in the pushdown query when no outer joins are used. If an outer join is used anywhere in the join tree, there will be a tree of IJoins with a single root. This latter form is the ANSI perfered style. If you wish all pushdown queries containing joins to be in ANSI style have the ConnectorCapability.useAnsiJoin return true. See *Command Form Capabilities* for more.

3.1.4. IQueryCommand Structure

IQueryCommand (refered to in SQL as a Query Expression) is the base for both queries and set queries. It may optionally take an IOrderBy (representing a SQL ORDER BY clause) and a ILimit (represent a SQL LIMIT clause)

3.1.5. IQuery Structure

Each IQuery will have an ISelect describing the expressions (typically elements) being selected and an IFrom specifying the group or groups being selected from, along with any join information. The IQuery may optionally also supply an ICriteria (representing a SQL WHERE clause), an IGroupBy (representing a SQL GROUP BY clause), an an ICriteria (representing a SQL HAVING clause).

3.1.6. ISetQuery Structure

An ISetQuery represents on of the SQL set operations (UNION, INTERSECT, EXCEPT) on two IQueryCommands. The all flag may be set to indicate UNION ALL (currently INTERSECT and EXCEPT ALL are not allowed in Teiid)

3.1.7. IInsert Structure

Each IInsert will have a single IGroup specifying the group being inserted into. It will also a list of IElements specifying the columns of the IGroup that are being inserted into and an IInsertValueSource, which will either be a list of IExpression (IInsertExpressionValueSource) or an IQueryCommand.

3.1.8. IUpdate Structure

Each IUpdate will have a single IGroup specifying the group being updated. The ISetClauseList contains ISetClause entries that specify IElement and IExpression pairs for the update. The IUpdate may optionally provide a criteria specifying which rows should be updated.

3.1.9. IDelete Structure

Each IDelete will have a single IGroup specifying the group being deleted from. It may also optionally have a criteria specifying which rows should be deleted.

3.1.10. IProcedure Structure

Each IProcedure has zero or more IParameter objects. The IParameter objects describe the input parameters, the output result set, and the output parameters.

3.1.11. IBatchedUpdate Structure

Each IBatchedUpdate has a list of ICommand objects (which must be an IInsert, IUpdate, or IDelete) that compose the batch.

3.2. Language Utilities

This section covers utilities available when using, creating, and manipulating the language interfaces.

3.2.1. Data Types

The Connector API contains an interface TypeFacility that defines data types and provides value translation facilities.

This ConnectorEnvironment (provided by Teiid on connector start) is a factory to obtain a TypeFacility instance for the connector using the getTypeFacility() method. The TypeFacitlity interface has methods that support data type transformation and detection of appropriate runtime or JDBC types. The TypeFacility.RUNTIME_TYPES and TypeFacility.RUNTIME_NAMES interfaces defines constants for all Teiid runtime data types. All IExpression instances define a data type based on this set of types. These constants are often needed in understanding or creating language interfaces.

3.2.2. Language Manipulation

In connectors that support a fuller set of capabilities (those that generally are translating to a language of comparable to SQL), there is often a need to manipulate or create language interfaces to move closer to the syntax of choice. Some utilities are provided for this purpose:

Similar to the TypeFacility, you can use the ConnectorEnvironment to get a reference to the ILanguageFactory instance for your connector. This interface is a factory that can be used to create new instances of all the concrete language interface objects.

Some helpful utilities for working with ICriteria objects are provided in the LanguageUtil class. This class has methods to combine ICriteria with AND or to break an ICriteria apart based on AND operators. These utilities are helpful for breaking apart a criteria into individual filters that your connector can implement.

3.3. Runtime Metadata

Teiid uses a library of metadata, known as "runtime metadata” for each virtual database that is deployed in Teiid. The runtime metadata is a subset of metadata as defined by models in the Teiid models that compose the virtual database.

Connectors can access runtime metadata by using the interfaces defined in org.teiid.connector.metadata.runtime. This package defines interfaces representing a MetadataID, a MetadataObject, and ways to navigate those IDs and objects.

3.3.1. Language Objects

One language interface, IMetadataReference describes whether a language object has a reference to a MetadataObject. The following interfaces extend IMetadataReference:

- IElement
  - returns an Element MetadataObject
- IGroup
  - returns a Group MetadataObject
- IProcedure
  - returns a Procedure MetadataObject
- IParameter
  - returns a Parameter MetadataObject

Once a MetadataObject has been obtained, it is possible to use it metadata about that object or to find other related or objects.

3.3.2. Access to Runtime Metadata

As mentioned in the previous section, a MetadataID is obtained from one of the language objects. That MetadataID can then be used directly to obtain information about the ID, such as the full name or short name.

The RuntimeMetadata interface is passed in for the creation of an Execution. It provides the ability to look up MetadataObjects based on their fully qualified names in the VDB. There are several kinds of MetadataObjects and they can be used to find more information about the object in runtime metadata.

Currently, only a subset of the most commonly used runtime metadata is available through these interfaces. In the future, more complete information will be available.

Obtaining MetadataObject Properties Example

The process of getting an element's properties is sometimes needed for connector development. For example to get the NameInSource property or all extension properties:

```
//getting the Group metadata from an IGroup is straight-forward
IGroup igroup = ... //some group on a command
Group group = igroup.getMetadataObject();

//we could also use the runtime metadata
RuntimeMetadata rm = ... //Obtained from the creation of the Execution

group = rm.getGroup("fully.qualified.name");
String contextName = group.getNameInSource();

//The props will contain extension properties
Properties props = group.getProperties();
```

3.4. Language Visitors

3.4.1. Framework

The Connector API provides a language visitor framework in the org.teiid.connector.visitor.framework package. The framework provides utilities useful in navigating and extracting information from trees of language objects.

The visitor framework is a variant of the Visitor design pattern, which is documented in several popular design pattern references. The visitor pattern encompasses two primary operations: traversing the nodes of a graph (also known as iteration) and performing some action at each node of the graph. In this case, the nodes are language interface objects and the graph is really a tree rooted at some node. The provided framework allows for customization of both aspects of visiting.

The base LanguageObjectVisitor class defines the visit methods for all leaf language interfaces that can exist in the tree. The LanguageObject interface defines an acceptVisitor() method – this method will call back on the visit method of the visitor to complete the contract. A base class with empty visit methods is provided as AbstractLanguageVisitor. The AbstractLanguageVisitor is just a visitor shell – it performs no actions when visiting nodes and does not provide any iteration.

The HierarchyVisitor provides the basic code for walking a language object tree. The HierarchyVisitor performs no action as it walks the tree – it just encapsulates the knowledge of how to walk it. If your connector wants to provide a custom iteration that walks the objects in a special order (to exclude nodes, include nodes multiple times, conditionally include nodes, etc) then you must either extend HierarchyVisitor or build your own iteration visitor. In general, that is not necessary.

The DelegatingHierarchyVisitor is a special subclass of the HierarchyVisitor that provides the ability to perform a different visitor's processing before and after iteration. This allows users of this class to implement either pre- or post-order processing based on the HierarchyVisitor. Two helper methods are provided on DelegatingHierarchyVisitor to aid in executing pre- and post-order visitors.

3.4.2. Provided Visitors

The SQLStringVisitor is a special visitor that can traverse a tree of language interfaces and output the equivalent Teiid SQL. This visitor can be used to print language objects for debugging and logging. The SQLStringVisitor does not use the HierarchyVisitor described in the last section; it provides both iteration and processing type functionality in a single custom visitor.

The CollectorVisitor is a handy utility to collect all language objects of a certain type in a tree. Some additional helper methods exist to do common tasks such as retrieving all elements in a tree, retrieving all groups in a tree, and so on.

3.4.3. Writing a Visitor

Writing your own visitor can be quite easy if you use the provided facilities. If the normal method of iterating the language tree is sufficient, then just follow these steps:

Create a subclass of AbstractLanguageVisitor. Override any visit methods needed for your processing. For instance, if you wanted to count the number of elements in the tree, you need only override the visit(IElement) method. Collect any state in local variables and provide accessor methods for that state.

Decide whether to use pre-order or post-order iteration. Note that visitation order is based upon syntax ordering of SQL clauses - not processing order.

Write code to execute your visitor using the utility methods on DelegatingHierarchyVisitor:

```
// Get object tree
ILanguageObject objectTree = …

// Create your visitor initialize as necessary
MyVisitor visitor = new MyVisitor();

// Call the visitor using pre-order visitation
DelegatingHierarchyVisitor.preOrderVisit(visitor, objectTree);

// Retrieve state collected while visiting
int count = visitor.getCount();
```

3.5. Connector Capabilities

All connectors must return a ConnectorCapabilities class from the `Connection.getCapabilities()` or `Connector.getCapabilities()` method. This class is used by the Connector Manager to determine what kinds of commands the connector is capable of executing. A basic implementation of the ConnectorCapabilities interface is supplied at BasicConnectorCapabilities. This capabilities class specifies that the connector does not support any capability. You should extend this class and override the necessary methods to specify which capabilities your connector supports.

3.5.1. Capability Scope

Note that if your capabilities will remain unchanged for the lifetime of the connector, you should return them via `Connector.getCapabilities()` since the engine will cache them for reuse by all connections to the connector. Capabilities returned by the connection will only be cached for the duration of the user request.

3.5.2. Capabilities

The following table lists the capabilities that can be specified in the ConnectorCapabilities class.

Table 3.1. Available Connector Capabilities

Capability	Requires	Description
SelectDistinct		Connector can support SELECT DISTINCT in queries.
SelectExpression		Connector can support SELECT of more than just element references.
AliasedGroup		Connector can support groups in the FROM clause that have an alias.
SupportedJoinCriteria	At least one of the join type supports.	Returns one of the SupportedJoinCriteria enum types: ANY, THETA, EQUI, KEY. KEY is the most restrictive, indicating that the source only supports equi-join criteria specified on the primary key of at least one of the tables in join.
InnerJoins		Connector can support inner and cross joins
SelfJoins	AliasedGroups and at least on of the join type supports.	Connector can support a self join between two aliased versions of the same group.
OuterJoins		Connector can support LEFT and RIGHT OUTER JOIN.
FullOuterJoins		Connector can support FULL OUTER JOIN.
InlineViews	AliasedGroup	Connector can support a named subquery in the FROM clause.
BetweenCriteria		Not currently used - between criteria is rewriten as compound comparisions.
CompareCriteriaEquals		Connector can support comparison criteria with the operator "=”.
CompareCriteriaOrdered		Connector can support comparison criteria with the operator ">” or "<".
LikeCriteria		Connector can support LIKE criteria.
LikeCriteriaEscapeCharacter	LikeCriteria	Connector can support LIKE criteria with an ESCAPE character clause.
InCriteria		

Capability	Requires	Description
		Connector can support IN predicate criteria.
InCriteriaSubquery		Connector can support IN predicate criteria where values are supplied by a subquery.
IsNullCriteria		Connector can support IS NULL predicate criteria.
OrCriteria		Connector can support the OR logical criteria.
NotCriteria		Connector can support the NOT logical criteria. IMPORTANT: This capability also applies to negation of predicates, such as specifying IS NOT NULL, "<=" (not ">"), ">=" (not "<"), etc.
ExistsCriteria		Connector can support EXISTS predicate criteria.
QuantifiedCompareCriteriaAll		Connector can support a quantified comparison criteria using the ALL quantifier.
QuantifiedCompareCriteriaSome		Connector can support a quantified comparison criteria using the SOME or ANY quantifier.
OrderBy		Connector can support the ORDER BY clause in queries.
OrderByUnrelated	OrderBy	Connector can support the ORDER BY items that are not directly specified in the select clause.
GroupBy		Connector can support an explict GROUP BY clause.
Having	GroupBy	Connector can support the HAVING clause.
AggregatesAvg		Connector can support the AVG aggregate function.
AggregatesCount		Connector can support the COUNT aggregate function.
AggregatesCountStar		Connector can support the COUNT(*) aggregate function.

Capability	Requires	Description
AggregatesDistinct	At least one of the aggregate functions.	Connector can support the keyword DISTINCT inside an aggregate function. This keyword indicates that duplicate values within a group of rows will be ignored.
AggregatesMax		Connector can support the MAX aggregate function.
AggregatesMin		Connector can support the MIN aggregate function.
AggregatesSum		Connector can support the SUM aggregate function.
ScalarSubqueries		Connector can support the use of a subquery in a scalar context (wherever an expression is valid).
CorrelatedSubqueries	At least one of the subquery pushdown capabilities.	Connector can support a correlated subquery that refers to an element in the outer query.
CaseExpressions		Not currently used - simple case is rewriten as searched case.
SearchedCaseExpressions		Connector can support "searched" CASE expressions anywhere that expressions are accepted.
Unions		Connector support UNION and UNION ALL
Intersect		Connector supports INTERSECT
Except		Connector supports Except
SetQueryOrderBy	Unions, Intersect, or Except	Connector supports set queries with an ORDER BY
RowLimit		Connector can support the limit portion of the limit clause
RowOffset		Connector can support the offset portion of the limit clause
FunctionsInGroupBy	GroupBy	Not currently used - non-element expressions in the group by create an inline view.
InsertWithQueryExpression		Connector supports INSERT statements with values specified by an IQueryCommand.

Note that any pushdown subquery must itself be compliant with the connector capabilities.

3.5.3. Command Form

The method ConnectorCapabilities.useAnsiJoin() should return true if the Connector prefers the use of ANSI style join structure for join trees that contain only INNER and CROSS joins.

The method ConnectorCapabilities.requiresCriteria() should return true if the Connector requires criteria for any Query, Update, or Delete. This is a replacement for the model support property "Where All".

3.5.4. Scalar Functions

The method ConnectorCapabilities.getSupportedFunctions() can be used to specify which scalar functions the connector supports. The set of possible functions is based on the set of functions supported by Teiid. This set can be found in the *Reference* [http://www.jboss.org/teiid//docs.html] documentation. If the connector states that it supports a function, it must support all type combinations and overloaded forms of that function.

There are also five standard operators that can also be specified in the supported function list: +, -, *, /, and ||.

The constants interface SourceSystemFunctions contains the string names of all possible built-in pushdown functions. Note that not all system functions appear in this list. This is because some system functions will always be evaluted in Teiid, are simple aliases to other functions, or are rewriten to a more standard expression.

3.5.5. Physical Limits

The method ConnectorCapabilities.getMaxInCriteriaSize() can be used to specify the maximum number of values that can be passed in an IN criteria. This is an important constraint as an IN criteria is frequently used to pass criteria between one source and another using a dependent join.

The method ConnectorCapabilities.getMaxFromGroups() can be used to specify the maximum number of FROM Clause groups that can used in a join. -1 indicates there is no limit.

3.5.6. Update Execution Modes

The method ConnectorCapabilities.supportsBatchedUpdates() can be used to indicate that the connector supports executing the IBatchedUpdates command.

The method ConnectorCapabilities.supportsBulkUpdate() can be used to indicate that the connector accepts update commands containg multi valued ILiterals.

Note that if the connector does not support either of these update modes, the query engine will compensate by issuing the updates individually.

Using the Connector Development Kit

4.1. Overview

The Connector Developer Kit (CDK) is a set of programmatic and command line utilities for testing connectors. The programmatic components of the CDK are useful for unit testing your connector and the command line utilities is useful for integration testing and regression testing (due to scripting abilities).

This chapter covers usage of both aspects of the CDK. For more detailed information about the CDK programmatic utilities also consult the *Teiid JavaDocs* [http://docs.jboss.org/teiid/6.2.0/apidocs].

4.2. Programmatic Utilities

All components provided by the CDK are in the package com.metamatrix.cdk.api.

4.2.1. Language Translation

Commands are sent to the Connector API in terms of the language interfaces discussed in the *Command Language* chapter. Typically, a connector must write logic to read and sometimes manipulate these objects. The CDK language translation utilities can be used to write unit tests for translation code or command execution.

The utilities are provided in the class TranslationUtility. This class has the following methods:

Table 4.1. Language Translation

Method Name	Description
TranslationUtility(String vdbFile)	Constructor – takes the path to a file which is a valid metadata archive created by the Teiid Designer. These files have the suffix ".vdb".
createRuntimeMetadata()	Creates an instance of RuntimeMetadata that can be used to test code that uses runtime metadata when translating or executing commands.
parseCommand(String sql)	Take a single-source command and return an ICommand that can be used to test translation or execution of commands.

4.2.2. Command Execution

The primary purpose of a Connector is to execute commands against an information source. The query execution utilities allow you to test the execution of commands programmatically. This

utility does not run the Teiid query engine or the connector manager although does simulate what happens when those components use a Connector to execute a command.

The command execution utilities are provided in the class ConnectorHost. This class has the following methods:

Table 4.2. Command Execution

Method Name	Description
ConnectorHost	Constructor – takes a Connector instance, a set of connector property values, and the path to a VDB archive file
setBatchSize	Sets the batch size to use when executing commands.
setExecutionContext	Sets the security context values currently being used to execute commands. This method may be called multiple times during the use of a single instance of ConnectorHost to change the current context.
getConnectorEnvironmentProperties	Helper method to retrieve the properties passed to the ConnectorHost constructor.
executeCommand	Execute a command and return the results using this connector.
executeBatchedUpdates	Execute a set of commands as a batched update.
getCommand	Use the host metadata to get the ICommand for a SQL string.

Here is some example code showing how to use ConnectorHost to test a connector:

```
// Prepare state for testing
MyConnector connector = new MyConnector();
Properties props = new Properties();
props.setProperty(“user”, “myuser”);
props.setProperty(“password”, “mypassword”);
String vdbFile = “c:/mymetadata.vdb”;

// Create host
ConnectorHost host = new ConnectorHost(connector, props, vdbFile);

// Execute query
List results = host.executeCommand(“SELECT col FROM group WHERE col = 5”);
```

```
// Compare actual results to expected results
// . . .
```

The executeCommand() method will return results as a List of rows. Each row is itself a List of objects in column order. So, each row should have the same number of items corresponding to the columns in the SELECT clause of the query. In the case of an INSERT, UPDATE, or DELETE, a single “row” will be returned with a single column that contains the update count.

4.3. Connector Environment

Many parts of the Connector API require use of the Connector Environment. The EnvironmentUtility can be used to obtain and control a Connector Environment instance.

Table 4.3. Command Execution

Method Name	Description
createExecutionContext	Creates a ExecutionContext instance.
createStdoutLogger	Creates an instance of ConnectorLogger that prints log messages to system.out( )
createEnvironment	Creates an instance of connectorEnvironment for use in your testing environment.
createExecutionContext	Creates an ExecutionContext instance.

In addition, some implementations of ConnectorLogger are provided which can be used as needed to build a custom logger for testing. BaseLogger is a base logger class that can be extended to create your own ConnectorLogger implementation. SysLogger is a utility implementation that logs to System.out.

4.4. Command Line Tester

4.4.1. Using the Command Line Tester

The command line tester is available in the mmtools kit along with the other Teiid products in the tools directory. The tester can be executed in interactive mode by running

```
<unzipped folder>S\cdk\cdk.bat
```

Typing “help” in the command line tester provides a list of all available options. These options are listed here with some additional detail:

Table 4.4. Connector Lifecycle

Option	Arguments	Description
Load Archive	ArchiveFileName	Load the Connector archive file, which loads the Connector type definition file and all the extension modules into the CDK shell.
Load	ConnectorClass vdbFile	Load a connector by specifying the connector class name and the VDB metadata archive file
LoadFromScript	ScriptFile	Load a connector from a script
LoadProperties	PropertyFile	Load a set of properties for your connector from a file
SetProperty	PropertyName PropertyValue	Set the value of a property
GetProperties		List all properties currently set on the connector
Start		Start the connector
Stop		Stop the connector

Table 4.5. Command Execution

Option	Arguments	Description
Select	Sql	Run a SELECT statement. This option takes multi-line input terminated with “;”
Insert	Sql	Execute an INSERT statement. This option takes multi-line input terminated with a “;”.
Update	Sql	Execute an UPDATE statement. This option takes multi-line input terminated with “;”
Delete	Sql	Execute a DELETE statement. This option takes multi-line input terminated with a “;”.
SetBatchSize	BatchSize	Set the batch size used when retrieving results
SetExecutionContext	VDBName VDBVersion UserName	Set the properties of the current security context
SetPrintStackOnError	PrintStackOnError	Set whether to print the stack trace when an error is received

Table 4.6. Scripting

Option	Arguments	Description
SetScriptFile	ScriptFile	Set the script file to use
Run	ScriptName	Run a script with the file name
Runall		Run all scripts loaded by loadFromScript
RunScript	ScriptFile ScriptNameWithinFile	Run a particular script in a script file
SetFailOnError	FailOnError	Set whether to fail a script when an error is encountered or continue on
Result	ExpectedResults	Compares actual results from the previous command with the expected results. This command is only available when using the command line tester in script mode.

Table 4.7. Miscellaneous

Option	Arguments	Description
CreateArchive	ArchiveFileName CDKFileName ExtensionModuleDir	Creates a connector archive file based on the properties supplied.
CreateTemplate	TemplateFile	Create a template connector type file at the given file name.
Help		List all options
Quit		Quit the command line tester

4.4.2. Loading Your Connector

Preparing your connector to execute commands consists of the following steps:

1. Add your connector code to the CDK classpath. The cdk.bat script looks for this code in the CONNECTORPATH environment variable. This variable can be set with the DOS shell command “SET CONNECTORPATH=c:\path\to\connector.jar”. Alternately, you can modify the value of the CONNECTORPATH environment variable in the cdk.bat file.

2. Start the command line tester. You can start the tester by executing the cdk.bat file in the cdk directory of the Teiid Tools installation.

3. Load your connector class and the associated runtime metadata. You can load your connector by using the “load” command and specifying the fully-qualified class name of your Connector

implementation and the path to a VDB file. The VDB runtime metadata archive should contain the metadata you want to use while testing.

4. Set any properties required by your connector. This can be accomplished with the setProperty command for individual properties or the loadProperties command to load a set of properties from either a properties file or a connector binding file. You can use the “getProperties” command to view the current property settings.

5. Start the connector. Use the “start” command in the command-line tester to start your connector.

Following is an example transcript of how this process might look in a DOS command window. User input is in bold.

```
D:\teiid\cdk> set CONNECTORPATH=D:\myconn\myconn.jar
D:\teiid\cdk> cdk.bat
========================== ENV SETTINGS ==========================
TEIID_ROOT     = D:\teiid
CONNECTORPATH  = D:\myconn\myconn.jar
CLASSPATH      = ;D:\teiid\cdk\metamatrix-cdk.jar;D:\myconn\myconn.jar;
==================================================================

java -Xmx256m com.metamatrix.cdk.ConnectorShell
Starting
Started
>load com.metamatrix.myconn.MyConnector d:\myconn\myconn.vdb
>setproperty user joe
>start
>
```

4.4.3. Executing Commands

Commands can be executed against your connector using the SELECT, INSERT, UPDATE, and DELETE commands. Procedure execution is not currently supported via the command line tester. Commands may span multiple lines and should be terminated with a “;”.

When a command is executed, the results are printed to the console. Following is an example session executing a SELECT command with the command line tester. User input is in bold.

```
>SELECT Name, Value FROM MyModel.MyGroup WHERE Name = ‘xyz’;
String Integer
```

```
xyz 5
xyz 10
>
```

4.4.4. Scripting

One of the most useful capabilities of the command-line tester is the ability to capture a sequence of commands in a script and automate the execution of the script. This allows for the rapid creation of regression and acceptance tests.

A script file may contain multiple scripts, where each script is grouped together with { } and a name. Following is an example of a script file. This script file also uses the special script-only command RESULTS that will compare the results of the last execution with the specified expected results.

```
test {
  load com.metamatrix.myconn.MyConnector d:\myconn\myconn.vdb
  setproperty user joe
  start

  SELECT Name, Value FROM MyModel.MyGroup WHERE Name = 'xyz';
  results [
  String Integer
  xyz 5
  xyz 10
  ]
}
```

To execute this file, run the command line tester in scripting mode and specify the script file and the script within the file:

```
D:\teiid\cdk>cdk runscript d:\myconn\my.script test
========================= ENV SETTINGS =========================
TEIID_ROOT     = D:\teiid
CONNECTORPATH  = D:\myconn\myconn.jar
CLASSPATH      = ;D:\teiid\cdk\metamatrix-cdk.jar;D:\myconn\myconn.jar;
================================================================
```

```
java -Xmx256m -Dmetamatrix.config.none -Dmetamatrix.log=4
 com.metamatrix.cdk.ConnectorShell runscript my.script
Starting
Started
>Executing: load com.metamatrix.myconn.MyConnector d:\myconn\myconn.vdb
>Executing: setproperty user joe
>Executing: start
>Executing: select Name, Value from MyModel.MyGroup where Name = 'xyz';
String Integer
xyz 5
xyz 15

>Test /teiid/cdk/yahoo.script.test failed.  CompareResults Error: Value mismatch at row 2 and
 column 2: expected = 10, actual = 15

>Finished
D:\teiid\cdk>
```

The script run above illustrates the output when the test result fails due to differences between expected and actual results. In this case the value was expected to be 10 in the script but was actually 15. The setFailOnError command can be used to fail the execution of the entire script if an error occurs.

Scripts can also be run in interactive mode by using the setScriptFile and run commands. This can be useful to record portions of your interactive testing to avoid re-typing later.

Connector Deployment

5.1. Overview

Once you have written and compiled the code for your connector, there are several steps to deploy your connector to Teiid:

- Creating a Connector Type Definition file that defines the properties required to initialize your connector.
- Identifying the Extension Modules (jars and resources) required for the Connector to run.
- Creating the Connector Archive file to bundle the Connector Type Definition file and the Extension Modules.
- Creating a Connector Binding using your Connector Type.

This chapter will help you perform these steps.

5.2. Connector Type Definition File

A Connector Type Definition file defines a connector in Teiid. The Connector Type Definition file defines some key properties that allow Teiid to use your connector as well as specifying other properties your connector might need.

A Connector Type Definition file is in XML format and typically has the extension ".cdk". It defines a default name for the connector type, the properties expected by the connector, and other information that allows the properties to be displayed correctly in the Console when a Connector Binding is created from the Connector Type.

An example of this file can be found in *Appendix A.* It may be helpful to refer to this file while reading this section. The template file can also be created using the Connector Development Kit.

5.2.1. Connector Binding Properties

The Connector API has built-in mechanisms for using the properties defined in the Connector ComponentType definition in the configuration.xml located in your deploy directory. For custom connectors the following properties are of primary importance:

Table 5.1. Connector Properties

Property Name	Example Value	Description
ConnectorClass	foo.MyConnector	Fully-qualified name of class implementing the Connector interface.
ConnectorClassPath	extensionjar:foo.jar	Semi-colon delimited list of jars defining the classpath of this connector. Typically this

Property Name	Example Value	Description
		includes the actual code for your connector as well as any 3rd party dependencies.

For more information on the Connector Classpath, see the section *Understanding the Connector Classpath*

5.2.2. Connector Properties

Most connectors require some initialization parameters to connect to the underlying enterprise information system. These properties can be defined in the Connector Type Definition file along with their default values and other property metadata. The actual property values can be changed when the connector is deployed in the Teiid Console.

Each connector property carries with it several attributes that are used by the Teiid Console to integrate the connector seamlessly into Teiid.

Table 5.2. All Attributes

Attribute Name	Example Value	Description
Name	ExampleProperty	Property name – should only contain letters, no spaces or other punctuation. This is the name of the property as it will be passed to the connector in the ConnectorEnvironment.
DisplayName	Example property	The property name as displayed in the Console. Typically this is a nicely formatted version of the Name attribute.
ShortDescription	The example property is used to control something.	A short description that is displayed as a tooltip of the property in the Teiid Console.
DefaultValue	Xyz	A default value for the property. This value will be auto-filled when a connector binding is created from the Connector Type.
IsRequired	false	If true, then this property is required. Any required property without a value is displayed in red in the connector binding properties panel.
IsModifiable	true	If set to “false”, the property is visible only when viewing all properties and is not modifiable in the properties panel.
IsMasked	false	If set to “true”, the property will be masked with *’s when it is entered and saved in an encrypted form. This attribute is typically used with passwords.

Attribute Name	Example Value	Description
IsExpert	true	Depending on the property display, the property can be optionally displayed for advanced users.
PropertyType	String	The short name of a built-in Java primitive wrapper Object type. Other possible values include Integer, Boolean, etc.

A property may also be constrained to a set of allowed values by adding child AllowedValue elements, i.e. <AllowedValue>value</AllowedValue>. Adding allowed values will cause the property to be displayed with a dropdown that limits the user selection to the allowed values.

5.3. Extension Modules

Extension Modules are used in Teiid to store code that extends Teiid in a central managed location. Extension Module JAR files are stored in the repository database and all Teiid processes access this database to obtain extension code. Custom connector code is typically deployed as extension models.

5.3.1. Understanding the Connector Classpath

By default each connector binding is loaded using the Teiid common classloader. Any needed extension modules are automatically added to common classpath. The common classloader is also a delegating classloader, so it's possible for classes to be found from the classpath set for Teiid or it's containing application.

If class conflicts would arise from delegation or shared classloading, each connector binding can be loaded in an isolated classloader (shared only by connectors with the same classpath), by setting the connector binding property UsePostDelegation to true. This classloading mode loads classes via the Teiid Extension Modules before loading classes from higher level classloaders.

The ConnectorClasspath property of your connector defines the extension module jars that are included in your connector’s classpath. The connector classpath is defined as a semi-colon delimited list of extension modules. Extension module jar files must be prefixed with "extensionjar:"

5.4. Connector Archive File

The Connector Archive file is a bundled version of all files needed by this Connector to execute in Teiid. This file includes the Connector Type Definition file and all the Extension Modules required by the Connector to create a connector archive file (CAF)..

- The archive is a standard zip file.
- Start the CDK tool by executing cdk.bat

- Execute “CreateArchive” command by supplying:

  1. Path to the name of the archive file to create
  2. Path to the Connector Type Definition file
  3. Path to the directory where the required Extension Modules (jar files) are stored (note that only .jar files specified in the ConnectorClassPath property of the Connector Type definition file are bundled.

The file created by the CDK can be opened with any zip file utility to verify the required files are included.

The archive file can be tested in the CDK tool by loading it using the command “loadArchive”. Refer *CDK chapter* for more information.

5.5. Importing the Connector Archive

5.5.1. Into Teiid

To use a new connector type definition in Teiid, the Connector Archive file must be imported via the AdminAPI via the addConnectorArchive method.

5.5.2. Into Teiid Designer

To use the new connector type during the development of the VDB for testing using the SQLExplorer, Connector Archive File must be imported into the Designer tools. To perform this task, perform the following steps.

1. Start Designer
2. Open the project and in the “vdb” execute panel, click on the “Open the Configuration Manager” link. For more information consult the designer’s guide.

1. In the result window, click “Import a Connector Type (.cdk,.caf)” link and follow directions.

The Connector Type can now be used to create Connector Bindings.

5.6. Creating a Connector Binding

5.6.1. In Designer

Connector Binding properties can also be defined in the Designer for the given Connector Type, if the corresponding Connector Archive File is imported into the Designer. If you try to execute your VDB with SQLExplorer in the Designer, this tool will present you with a window to specify such Connector Bindings. The user is required specify these binding properties before they can

test using the SQLExplorer. For more information on how this can be accomplished please refer to the *Designer User's Guide* [http://www.jboss.org/teiiddesigner/docs.html].

Also, note that the bindings specified in the Designer tool are automatically bundled into the VDB for deployment, so if there are any properties that needs to be changed from development environment to the production environment, those properties need to be modified when a VDB is later deployed.

Connection Pooling

6.1. Overview

The Query Engine logically obtains and closes a connection for each command.

However many enterprise sources connections can be persistent and expensive to create. For these situations, Teiid provides a transparent connection pool to reuse, rather than constantly close, connections. The connection pool is highly configurable through configuration properties and extension APIs for Connections and Connectors

Many built-in connector types take advantage of pooling, including JDBC, Salesforce, and LDAP connectors.

6.2. Framework Overview

The table below lists the role of each class in the framework.

Table 6.1. Responsibilities of Connection Pool Classes

Class	Type	Description
Connection	Interface	The isAlive and closeCalled methods are used for pool interaction.
ConnectorIdentity	Interface	This interface corresponds to an identifier for a connection in the pool. Changing the identity implementation changes the basis on which connections are pooled. Connections that have equal identity objects (based on the equals() method) will be in the same pool.
SingleIdentity	Class	This implementation of ConnectorIdentity makes all connections equivalent, thus user scoping of connection is ignored.
MappedUserIdentity	Class	This implementation of ConnectorIdentity makes all connections equivalent for a particular user allowing for per-user connection pools.
ConnectionPooling	Annotation	This optional Annotation can be used on the Connector implementation class to indicate configure pooling. This can be especially useful to indicate that automatic ConnectionPooing should not be used regardless of the connector binding property settings.

6.3. Using Connection Pooling

Automatic connection pooling does not require any changes to basic Connector development. It can be enabled by setting the Connector binding Property ConnectionPoolEnabled=true or by adding the ConnectionPooling annotation, which defaults to enabled=true, to the Connector implementation class. Automatic Connection pooling can be disabled if either setting is false.

It is important to consider providing an implementation for Connection.isAlive to indicate that a Connection is no longer viable and should be purged from the pool. Connection testing is performed upon leases from the pool and optionally at a regular interval that will purge idle Connections. It is also important to consider having the concrete Connector class implement ConnectorIdentity factory if Connections are made for multiple identities. Note that setting connector binding property UseCredentialMap to true will allow connectors extending BasicConnector to have their ConnectorIdentity automatically set based upon the user CredentialMap.

6.4. The Connection Lifecycle

These steps occur when connection pooling is enabled:

1. The ConnectorManager asks the Connector to generate a ConnectorIdentity for the given ExecutionContext. The ConnectorIdentity is then stored on the ExecutionContext.
2. The ConnectorManager asks for a Connection from the pool that pertains to the ConnectorIdentity.
3. The ConnectionPool returns a Connection that was either pulled from the pool (and passes the isAlive check) or was created by the Connector if necessary.
4. After the ConnectorManager has used the Connection to execute a command, it releases the Connection. This call is intercepted by the pool and the method Connection.closeCalled is invoked on the Connection instead. Note that for many sources no action is necessary on closeCalled.
5. When the Connection fails an isAlive check or becomes too old with pool shrinking enabled, it is purged from the pool and Connection.close is called.

6.4.1. XAConnection Pooling

The usage of XAConnections (that provide XAResources) typically come with additional limitations about how those Connections can be used once they are enlisted in a transaction. When enabled, automatic connection pooling will perform these additional features with XAConnections:

- The pool will return the same XAConnection for all executions under a given transaction until that transaction completes. This implies that all executions to a given XAConnector under the same connection will happen serially.

- XAConnections enlisted in a transaction will return to the pool once a transaction completes.
- Two separate pools will be maintained. One for Connections that have not and will not be used in a transaction, and one for XAConnections that have an will be used in a transaction. Each pool will be configured based upon the same set of configuration properties - it is not possible to independently control pool sizes, etc.

6.5. Configuring the Connection Pool

The ConnectionPool has a number of properties that can be configured via the connector binding expert properties. Note *. indicates that the property prefix is com.metamatrix.data.pool.

Table 6.2. Connection Pool Properties

Name	Key	Default Value	Description
Connection Pool Enabled	ConnectionPoolEnabled		Explicitly enables or disables connection pooling.
Data Source Test Connect Interval (seconds)	SourceConnectionTestInterval	600	How often (in seconds) to create test connections to the underlying source to see if it is available.
Pool Maximum Connections	*.max_connections	20	Maximum number of connections total in the pool. This value should be greater than 0.
Pool Maximum Connections for Each ID	*.max_connections_for_each_id	20	Maximum number of connections per ConnectorIdentity object. This value should be greater than 0.
Pool Connection Idle Time (seconds)	*.live_and_unused_time	60	Maximum idle time (in seconds) before a connection is closed if shrinking is enabled.
Pool Connection Waiting Time (milliseconds)	*.wait_for_source_time	120000	Maximum time to wait (in milliseconds) for a connection to become available.
Pool cleaning Interval (seconds)	*.cleaning_interval	60	Interval (in seconds) between checking for idle connections if shrinking is enabled.
Enable Pool Shrinking	*.enable_shrinking	true	Indicate whether the pool is allowed to shrink.

Handling Large Objects

This chapter examines how to use facilities provided by the Teiid Connector API to use large objects such as blobs, clobs, and xml in your connector.

7.1. Large Objects

7.1.1. Data Types

Teiid supports three large object runtime data types: blob, clob, and xml. A blob is a “binary large object”, a clob is a “character large object”, and “xml” is a “xml document”. Columns modeled as a blob, clob, or xml are treated similarly by the connector framework to support memory-safe streaming.

7.1.2. Why Use Large Object Support?

Teiid allows a Connector to return a large object through the Teiid Connector API by just returning a reference to the actual large object. Access to that LOB will be streamed as appropriate rather than retrieved all at once. This is useful for several reasons:

1. Reduces memory usage when returning the result set to the user.
2. Improves performance by passing less data in the result set.
3. Allows access to large objects when needed rather than assuming that users will always use the large object data.
4. Allows the passing of arbitrarily large data values.

However, these benefits can only truly be gained if the Connector itself does not materialize an entire large object all at once. For example, the Java JDBC API supports a streaming interface for blob and clob data.

7.2. Handling Large Objects

The Connector API automatically handles large objects (Blob/Clob/SQLXML) through the creation of special purpose wrapper objects when it retrieves results.

Once the wrapped object is returned, the streaming of LOB is automatically supported. These LOB objects then can for example appear in client results, in user defined functions, or sent to other connectors.

A connector execution is usually closed and the underlying connection is either closed/released as soon as all rows for that execution have been retrieved. However, LOB objects may need to be read after their initial retrieval of results. When LOBs are detected the default closing behavior is prevented by setting a flag on the ExecutionContext.

Now the connector execution only when the user Statement object is closed. Note that connectors may at their discretion have executions delayed in their closure by directly setting the keep alive on the ExecutionContext

```
executionContext.keepExecutionAlive(true);
```

An important limitation of using the LOB type objects is that streaming is not supported from remote connectors. This is an issue in clustered environments if connectors intended to return LOBs are deployed on only a subset of the hosts or in failover situations. The most appropriate workaround to this limitation is to deploy connectors intended to return LOBs on each host in the cluster.

7.3. Inserting or Updating Large Objects

LOBs will be passed to the Connector in the language objects as an ILiteral containing a java.sql.Blob, java.sql.Clob, or java.sql.SQLXML. You can use these interfaces to retrieve the data in the large object and use it for insert or update.

Appendix A. Connector Type Definition Template

This appendix contains an example of the Connector Type Definition file that can be used as a template when creating a new Connector Type Definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<ConfigurationDocument>
   <Header>
      <ApplicationCreatedBy>Connector Development Kit</ApplicationCreatedBy>
      <ApplicationVersionCreatedBy>4.0:1681</ApplicationVersionCreatedBy>
      <UserCreatedBy>MetaMatrixAdmin</UserCreatedBy>
      <DocumentTypeVersion>1.0</DocumentTypeVersion>
      <MetaMatrixSystemVersion>4.0</MetaMatrixSystemVersion>
      <Time>2008-01-30T15:22:05.296-06:00</Time>
   </Header>
   <ComponentTypes>
                        <ComponentType Name="My Connector" ComponentTypeCode="2"
 Deployable="true" Deprecated="false" Monitorable="false" SuperComponentType="Connector"
 ParentComponentType="Connectors">
     <!-- Required by Connector API -->
         <ComponentTypeDefn Deprecated="false">
                                                <PropertyDefinition Name="ConnectorClass"
  DisplayName="Connector Class" ShortDescription="" DefaultValue="com.mycode.Connector"
 IsRequired="true" IsMasked="false" IsModifiable="false" />
         </ComponentTypeDefn>
         <ComponentTypeDefn Deprecated="false">
                                                <PropertyDefinition Name="ConnectorClassPath"
    DisplayName="Class Path" ShortDescription="" DefaultValue="extensionjar:mycode.jar"
 IsRequired="true" IsMasked="false"  />
         </ComponentTypeDefn>

     <!-- Example properties - replace with custom properties -->
         <ComponentTypeDefn Deprecated="false">
                  <PropertyDefinition Name="ExampleOptional" DisplayName="Example Optional
 Property" ShortDescription="This property is optional due to not being marked as IsRequired"
 IsMasked="false" />
         </ComponentTypeDefn>
         <ComponentTypeDefn Deprecated="false">
```

```
        <PropertyDefinition Name="ExampleDefaultValue" DisplayName="Example Default Value
 Property" ShortDescription="This property has a default value" DefaultValue="Default value"
IsRequired="true"  IsMasked="false" />
        </ComponentTypeDefn>
        <ComponentTypeDefn Deprecated="false">
            <PropertyDefinition Name="ExampleEncrypted" DisplayName="Example Encrypted
 Property" ShortDescription="This property is encrypted in storage due to Masked=true"
IsRequired="true" IsMasked="true" />
        </ComponentTypeDefn>

        <ChangeHistory>
          <Property Name="LastChangedBy">ConfigurationStartup</Property>
          <Property Name="CreatedBy">ConfigurationStartup</Property>
        </ChangeHistory>
      </ComponentType>
  </ComponentTypes>
</ConfigurationDocument>
```