Teiid - Scalable Information Integration

1

Teiid Developer's Guide

7.2

Developing For Teiid

1.1. Introduction to the Teiid Connector Architecture

Integrating data from a Enterprise Information System (EIS) into Teiid, is separated into two parts.

1. A Translator, which is required.
2. An optional Resource Adapter, which will typically be a JCA Resource Adapter (also called a JEE Connector)

A Translator is used to:

1. Translate a Teiid-specific command into a native command,
2. Execute the command,
3. Return batches of results translated to expected Teiid types.

A Resource Adapter:

- Handles all communications with individual enterprise information system (EIS), which can include databases, data feeds, flat files, etc.
- Can be a JCA Connector or any other custom connection provider. The reason Teiid recommends and uses JCA is this specification defines how one can write, package, and configure access to EIS system in consistent manner. There are also various commercial/open source software vendors already providing JCA Connectors to access a variety of back-end systems.

  Refer to *http://java.sun.com/j2ee/connector/*.
- Abstracts Translators from many common concerns, such as connection information, resource pooling, or authentication.

Given a combination of a Translator + Resource Adapter, one can connect any EIS system to Teiid for their data integration needs.

1.2. Do You Need a New Translator?

Teiid provides several translators 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 translators:

JDBC Translator
: Works with many relational databases. The JDBC translator is validated against the following database systems: Oracle, Microsoft SQL Server, IBM DB2, MySQL, Postgres, Derby, Sybase, H2, and HSQL. In addition, the JDBC Translator can often be used with other 3rd-party drivers and provides a wide range of extensibility options to specialize behavior against those drivers.

File Translator
: Provides a procedural way to access the file system to handle text files.

WS Translator
: Provides procedural access to XML content using Web Services.

LDAP Translator
: Accesses to LDAP directory services.

Salesforce Translator
: Works with Salesforce interfaces.

1.2.1. Custom Translators

Below are the high-level steps for creating custom Translators. This guide covers how to do each of these steps in detail. It also provides additional information for advanced topics, such as streaming large objects.

For sample Translator code, refer to the Teiid source code at *http://anonsvn.jboss.org/repos/teiid/trunk/connectors/*.

1. Create a new or reuse an existing Resource Adapater for the EIS system, to be used with this Translator.

   Refer to *Section 1.3.1, "Custom Resource Adapters"*.

2. Implement the required classes defined by the Translator API.

   - Create an ExecutionFactory – Extend the `org.teiid.translator.ExecutionFactory` class
   - Create relevant Executions (and sub-interfaces) – specifies how to execute each type of command

   Refer to *Chapter 3, Translator Development*.

3. Define the template for exposing configuration properties. Refer to *Section 3.6, "Packaging"*.
4. Deploy your Translator. Refer to *Section 3.7, "Deployment"*.
5. Deploy a Virtual Database (VDB) that uses your Translator.
6. Execute queries via Teiid.

1.3. Do You Need a New Resource Adapter?

As mentioned above, for every Translator that needs to gather data from external source systems, it requires a resource adapter.

The following resource adapters are available to Teiid.

- *DataSource:* This is provided by the JBoss AS container. This is used by the JDBC Translator.
- *File:* Provides a JEE JCA based Connector to access defined directory on the file system. This is used by the File Translator
- *WS:* Provides JEE JCA Connector to invoke Web Services using JBoss Web services stack. This is used by the WS Translator
- *LDAP:* Provides JEE JCA connector to access LDAP; Used by the LDAP Translator.
- *Salesforce:* Provides JEE JCA connector to access Salesforce by invoking their Web Service interface. Used by the SalesForce Translator.

1.3.1. Custom Resource Adapters

High-level Resource Adapter development procedure:

1. Understand the JEE Connector specification to have basic idea about what JCA connectors are how they are developed and packaged.

   Refer to *http://java.sun.com/j2ee/connector/*.

2. 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 processing constructs, or capabilities, supported by information system.
   - Required properties for the connection, such as URL, user name, etc.
3. Base classes for all of the required supporting JCA SPI classes are provided by the Teiid API. The JCA CCI support is not provided from Teiid, since Teiid uses the Translator API as it's common client interface. You will want to extend:
   - BasicConnectionFactory – Defines the Connection Factory
   - BasicConnection – represents a connection to the source.

- BasicResourceAdapter – Specifies the resource adapter class

4. Package your resource adapter. Refer to *Section 2.2, "Packaging the Adapter"*.

5. Deploy your resource adapter. Refer to *Section 2.2, "Packaging the Adapter"*.

This guide covers how to do each of these steps in detail. It also provides additional information for advanced topics, such as transactions. For sample resource adapter code refer to the Teiid Source code at *http://anonsvn.jboss.org/repos/teiid/trunk/connectors/*.

1.4. Other Teiid Development

Teiid is highly extensible in other ways:

- You may add User Defined Functions. Refer to *Chapter 5, User Defined Functions*.
- You may adapt logging to your needs, which is especially useful for custom audit or command logging. Refer to *Chapter 7, Logging*.
- You may change the subsystem for custom authentication and authorization. Refer to *Chapter 8, Login Modules*.

Developing JEE Connectors

This chapter examines how to use facilities provided by the Teiid API to develop a JEE JCA Connector. Please note that these are standard JEE JCA connectors, nothing special needs to be done for Teiid. As an aid to our Translator developers, we provided a base implementation framework. If you already have a JCA Connector or some other mechanism to get data from your source system, you can skip this chapter.

If you are not familiar with JCA API, please read the JCA 1.5 Specification at *http://java.sun.com/j2ee/connector/*. There are lot of online tutorials on how to design and build a JCA Connector. The below we show you to build very simple connector, however building actual connector that supports transactions, security can get much more complex.

Refer to the JBoss Application Server Connectors documentation at *http://docs.jboss.org/jbossas/jboss4guide/r4/html/ch7.chapt.html*.

2.1. Using the Teiid Framework

If you are going to use the Teiid framework for developing a JCA connector, follow these steps. The required classes are in `org.teiid.resource.api` package. Please note that Teiid framework does not make use JCA's CCI framework, only the JCA's SPI interfaces.

- Define Managed Connection Factory
- Define the Connection Factory class
- Define the Connection class
- Define the configuration properties in a "ra.xml" file

2.1.1. Define Managed Connection Factory

Extend the `BasicManagedConnectionFactory`, and provide a implementation for the "createConnectionFactory()" method. This method defines a factory method that can create connections.

This class also defines configuration variables, like user, password, URL etc to connect to the EIS system. Define an attribute for each configuration variable, and then provide both "getter" and "setter" methods for them. Note to use only "java.lang" objects as the attributes, DO NOT use Java primitives for defining and accessing the properties. See the following code for an example.

```
public class MyManagedConnectionFactory extends BasicManagedConnectionFactory
{
  @Override
  public Object createConnectionFactory() throws ResourceException
```

```
  {
    return new MyConnectionFactory();
  }

  // config property name (metadata for these are defined inside the ra.xml)
  String userName;
  public String getUserName()         {  return this.userName;  }
  public void setUserName(String name){  this.userName = name;  }

  // config property count  (metadata for these are defined inside the ra.xml)
  Integer count;
  public Integer getCount()           {  return this.count;  }
  public void setCount(Integer value) {  this.count = value; }

}
```

2.1.2. Define the Connection Factory class

Extend the `BasicConnectionFactory` class, and provide a implementation for the "getConnection()" method.

```
public class MyConnectionFactory extends BasicConnectionFactory
{
  @Override
  public MyConnection getConnection() throws ResourceException
  {
    return new MyConnection();
  }
}
```

Since the Managed connection object created the "ConnectionFactory" class it has access to all the configuration parameters, if "getConnection" method needs to do pass any of credentials to the underlying EIS system. The Connection Factory class can also get reference to the calling user's `javax.security.auth.Subject` during "getConnection" method by calling

```
Subject subject = ConnectionContext.getSubject();
```

This "Subject" object can give access to logged-in user's credentials and roles that are defined. Note that this may be null.

Note that you can define "security-domain" for this resource adapter, that is separate from the Teiid defined "security-domain" for validating the JDBC end user. However, it is users responsibility to

make the necessary logins before the Container's thread accesses this resource adapter, and this can get overly complex.

2.1.3. Define the Connection class

Extend the `BasicConnection` class, and provide a implementation based on your access of the Connection object in the Translator. If your connection is stateful, then override "isAlive()" and "cleanup()" methods and provide proper implementations. These are called to check if a Connection is stale or need to flush them from the connection pool etc. by the Container.

```
public class MyConnection extends BasicConnection
{
  public void doSomeOperation(command)
  {
    // do some operation with EIS system..
    // This is method you use in the Translator, you should know
    // what need to be done here for your source..
  }

  @Override
  public boolean isAlive()
  {
    return true;
  }

  @Override
  public void cleanUp()
  {

  }
}
```

2.1.4. XA Transactions

If your EIS source can participate in XA transactions, then on your Connection object, override the "getXAResource()" method and provide the "XAResource" object for the EIS system. Refer to *Section 2.1.3, "Define the Connection class"*. Also, You need to extend the "BasicResourceAdapter" class and provide implementation for method "public XAResource[] getXAResources(ActivationSpec[] specs)" to participate in crash recovery.

Note that, only when the resource adapters are XA capable, then Teiid can make them participate in a distributed transactions. If they are not XA capable, then source can participate in distributed query but will not participate in the transaction. Transaction semantics at that time defined by how you defined "-ds.xml" file. i.e. with local-tx or no-tx

2.1.5. Define the configuration properties in a "ra.xml" file

Define a "ra.xml" file in "META-INF" directory of your RAR file. An example file is provided in *Appendix A, ra.xml file Template*.

For every configuration property defined inside the ManagedConnectionFactory class, define the following XML configuration fragment inside the "ra.xml" file. These properties are used by user to configure instance of this Connector inside a Container. Also, during the startup the Container reads these properties from this file and knows how to inject provided values in the "-ds.xml" file into a instance of "ManagedConnectionFactory" to create the Connection. Refer to *Section 2.1.1, "Define Managed Connection Factory"*.

```
<config-property>
  <description>
    {$display:"${display-name}",$description:"${description}", $allowed="${allowed}",
    $required="${true|false}", $defaultValue="${default-value}"}
  </description>
  <config-property-name>${property-name}</config-property-name>
  <config-property-type>${property-type}</config-property-type>
  <config-property-value>${optioal-property-value}</config-property-value>
</config-property>
```

The format and contents of "<description>" element may be used as extended metadata for tooling. The special format must begin and end with curly braces e.g. {...}. This use of the special format and all properties is optional. Property names begin with '$' and are separated from the value with ':'. Double quotes identifies a single value. A pair of square brackets, e.g. [...], containing comma separated double quoted entries denotes a list value.

Extended metadata properties

- $display: Display name of the property
- $description: Description about the property
- $required: The property is a required property; or optional and a default is supplied
- $allowed: If property value must be in certain set of legal values, this defines all the allowed values
- $masked: The tools need to mask the property; Do not show in plain text; used for passwords
- $advanced: Notes this as Advanced property
- $editable: Property can be modified; or read-only

Note that all these are optional properties; however in the absence of this metadata, Teiid tooling may not work as expected.

2.2. Packaging the Adapter

Once all the required code is developed, it is time to package them into a RAR artifact, that can be deployed into a Conainer. A RAR artifact is lot more similar to a WAR. To put together a RAR file it really depends upon build system you are using.

- Eclipse: You can start out with building Java Connector project, it will produce the RAR file
- Ant: If you are using "ant" build tool, there is "rar" build task available
- Maven: If you are using maven, use <packaging> element value as "rar". Teiid uses maven, you can look at any of the "connector" projects for sample "pom.xml" file. Here is sample pom.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>connector-{name}</artifactId>
  <groupId>org.company.project</groupId>
  <name>Name Connector</name>
  <packaging>rar</packaging>
  <description>This connector is a sample</description>

  <dependencies>
    <dependency>
      <groupId>org.jboss.teiid</groupId>
      <artifactId>teiid-api</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.jboss.teiid</groupId>
      <artifactId>teiid-common-core</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.resource</groupId>
      <artifactId>connector-api</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>

</project>
```

Make sure that the RAR file, under its "META-INF" directory has the "ra.xml" file. If you are using maven refer to *http://maven.apache.org/plugins/maven-rar-plugin/*. In the root of the RAR file, you can embed the JAR file containing your connector code and any dependent library JAR files.

2.3. Deploying the Adapter

Once the RAR file is built, deploy it by copying the RAR file into "deploy" directory of JBoss AS's choosen profile. Typically the server does not need to be restarted when a new RAR file is being added. Alternatively, you can also use "admin-console" a web based monitoring and configuration tool to deploy this file into the container.

Once the Connector's RAR file is deployed into the JBoss container, now you can start creating a instance of this connector to be used with your Translator. Creating a instance of this Connector is no different than creating a "Connection Factory" in JBoss AS. Again, you have have two ways you can create a "ConnectionFactory".

- Create "${name}-ds.xml" file, and copy it into "deploy" directory of JBoss AS.

```
<!DOCTYPE connection-factories PUBLIC
       "-//JBoss//DTD JBOSS JCA Config 1.5//EN" "http://www.jboss.org/j2ee/dtd/jboss-ds_1_5.dtd">

<connection-factories>
    <no-tx-connection-factory>
      <jndi-name>${jndi-name}</jndi-name>
      <rar-name>${name}.rar</rar-name>
      <connection-definition>javax.resource.cci.ConnectionFactory</connection-definition>

  <!--
  define all the properties defined in the "ra.xml" that required or needs to be
  modified from defaults each property is defined in single element
  -->
            <config-property name="prop-name" type="java.lang.String">prop-value</config-property>

</no-tx-connection-factory>
</connection-factories>
```

  There are lot more properties that you can define for pooling, transactions, security etc in this file. Check JBoss AS documentation for all the avaialble properties.

- Alternatively you can use the web based "admin-console" configration and monitoring program, to create a new Connection Factory. Just have your RAR file name and needed configuration properties handly and fill out web form and create the ConnectionFactory.

Translator Development

3.1. Extending the ExecutionFactory Class

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

A custom translator must extend `org.teiid.translator.ExecutionFactory` to connect and query an enterprise data source. This extended class must provide a no-arg constructor that can be constructed using Java reflection libraries. This Execution Factory need define/override following elements.

3.1.1. ConnectionFactory

Defines the "ConnectionFactory" interface that is expected from resource adapter. This defined as part of class definition using generics while extending the "ExecutionFactory" class

3.1.2. Connection

Defines the "Connection" interface that is expected from resource adapter. This defined as part of class definition using generics while extending the "ExecutionFactory" class

3.1.3. Configuration Properties

Every software program requires some external configuration, that defines ways user can alter the behavior of a program. If this translator needs configurable properties define a variable for every property as an attribute in the extended "ExecutionFactory" class. Then define a "get" and "set" methods for each of them. Also, annotate each "get" method with `@TranslatorProperty` annotation and provide the metadata about the property.

For example, if you need a property called "foo", by providing the annotation on these properties, the Teiid tooling will automatically interrogate and provide graphical way to configure your Translator.

```
private String foo = "blah";
@TranslatorProperty(display="Foo property", description="description about Foo")
public String getFoo()
{
  return foo;
}

public void setFoo(String value)
{
```

```
  return this.foo = value;
}
```

Only java primitive (int), primitive object wrapper (java.lang.Integer), or Enum types are supported as Translator properties. The default value will be derived from calling the getter, if available, on a newly constructed instance. All properties *should* have a default value. If there is no applicable default, then the property should be marked in the annotation as required. Initialization will fail if a required property value is not provided.

The `@TranslatorProperty` defines the following metadata that you can define about your property

- display: Display name of the property
- description: Description about the property
- required: The property is a required property
- advanced: This is advanced property; A default should be provided. A property can not be "advanced" and "required" at same time.
- masked: The tools need to mask the property; Do not show in plain text; used for passwords

3.1.4. Initializing the Translator

Override and implement the `start` method (be sure to call "super.start()") if your translator needs to do any initializing before it is used by the Teiid engine. This method will be called by Teiid, once after all the configuration properties set above are injected into the class.

3.1.5. TranslatorCapabilities

These are various methods that typically begin with method signature "supports" on the "ExecutionFactory" class. These methods need to be overridden to describe the execution capabilities of the Translator. Refer to *Section 3.4.5, "Translator Capabilities"* for more on these methods.

3.1.6. Execution (and sub-interfaces)

Based on types of executions you are supporting, the following methods need to be overridden and need to provide implementations for these methods by extending respective interfaces.

- `createResultSetExecution` - Define if you are doing read based operation that is returning a rows of results.
- `createUpdateExecution` - Define if you are doing write based operations.

- `createProcedureExecution` - Define if you are doing procedure based operations.

You can choose to implement all the execution modes or just what you need. See more details on this below.

3.1.7. Metadata

Override and implement the method `getMetadata()`, if you want to expose the metadata about the source for use in Dynamic VDBs. This defines the tables, column names, procedures, parameters, etc. for use in the query engine. This method is not yet used by Designer tooling.

3.1.8. Logging

Teiid provides `org.teiid.logging.LogManager` class for logging purposes. Create a logging context and use the LogManager to log your messages. These will be automatically sent to the main Teiid logs. You can edit the "jboss-log4j.xml" inside "conf" directory of the JBoss AS's profile to add the custom context. Teiid uses Log4J as its underlying logging system.

3.1.9. Exceptions

If you need to bubble up any exception use `org.teiid.translator.TranslatorException` class.

3.1.10. Default Name

Finally, you can define a default instance of your Translator by defining the annotation `@Translator` on the "ExecutionFactory". When you define this, and after deployment a default instance of this Translator is available any VDB that would like to use by just mentioning its name in its "vdb.xml" configuration file. VDB can also override the default properties and define another instance of this Translator too. The name you give here is the short name used every where else in the Teiid configuration to refer to this translator.

3.2. Connections to Source

3.2.1. Obtaining connections

The extended "ExecutionFactory" must implement the `getConnection()` method to allow the Connector Manager to obtain a connection.

3.2.2. 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 closeConnection() method called on the "ExecutionFactory". You must also override this method to properly close the connection.

In cases (such as when a connection is stateful and expensive to create), connections should be pooled. If the resource adapter is JEE JCA connector based, then pooling is automatically

provided by the JBoss AS container. If your resource adapter does not implement the JEE JCA, then connection pooling semantics are left to the user to define on their own.

3.3. Executing Commands

3.3.1. Execution Modes

The Teiid query engine uses the "ExecutionFactory" class to obtain the "Execution" interface for the command it is executing. The actual queries themselves are sent to translators in the form of a set of objects, which are further described in Command Language. Refer to *Section 3.4, "Command Language"*. Translators are allowed to support any subset of the available execution modes.

Table 3.1. Types of Execution Modes

Execution Interface	Command interface(s)	Description
`ResultSetExecution`	`QueryExpression`	A query corresponding to a SQL SELECT or set query statement.
`UpdateExecution`	`Insert, Update, Delete, BatchedUpdates`	An insert, update, or delete, corresponding to a SQL INSERT, UPDATE, or DELETE command
`ProcedureExecution`	`Call`	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.

3.3.2. ResultSetExecution

Typically most commands executed against translators are QueryExpression. While the command is being executed, the translator provides results via the ResultSetExecution's "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.

3.3.3. Update Execution

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

3.3.4. 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.

3.3.5. Asynchronous Executions

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

Since the exection and the associated connection are not closed until the work has completed, care should be taken if using asynchronous executions that hold a lot of state.

3.3.6. Bulk Execution

Non batched `Insert`, `Update`, `Delete` commands may have `Literal` values marked as multiValued if the capabilities shows support for BulkUpdate. Commands with multiValued `Literal`s represent multiple executions of the same command with different values. As with BatchedUpdates, bulk operations should be executed atomically if possible.

3.3.7. 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 created in the Execution object.

3.3.8. 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.

3.4. Command Language

3.4.1. Language

Teiid sends commands to your Translator in object form. These classes are all defined in the "org.teiid.language" package. These objects can be combined to represent any possible command that Teiid may send to the Translator. However, it is possible to notify Teiid that your Translator can only accept certain kinds of constructs via the capabilities defined on the "ExecutionFactory" class. Refer to *Section 3.4.5, "Translator Capabilities"* for more information.

The language objects all extend from the `LanguageObject` 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 Translator are in the form of these language trees, where the root of the tree is a subclass of `Command`. Command has several sub-interfaces, namely:

- `QueryExpression`
- `Insert`
- `Update`
- `Delete`
- `BatchedUpdates`
- `Call`

Important components of these commands are expressions, criteria, and joins, which are examined in closer detail below. For more on the classes and interfaces described here, refer to the Teiid JavaDocs *http://docs.jboss.org/teiid/7.2/apidocs*.

3.4.1.1. 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 column reference such as "table.EmployeeName" represents a column in a data source and may take on many values while the command is being evaluated.

- `Expression` – base expression interface
- `ColumnReference` – represents an column in the data source

- `Literal` – represents a literal scalar value, but may also be multi-valued in the case of bulk updates.
- `Function` – represents a scalar function with parameters that are also Expressions
- `Aggregate` – represents an aggregate function which holds a single expression
- `ScalarSubquery` – represents a subquery that returns a single value
- `SearchedCase, SearchedWhenClause` – 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.

3.4.1.2. Condition

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.

- `Condition` – the base criteria interface
- `Not` – used to NOT another criteria
- `AndOr` – used to combine other criteria via AND or OR
- `SubuqeryComparison` – represents a comparison criteria with a subquery including a quantifier such as SOME or ALL
- `Comparison` – represents a comparison criteria with =, >, <, etc.
- `BaseInCondition` – base class for an IN criteria
- `In` – represents an IN criteria that has a set of expressions for values
- `SubqueryIn` – represents an IN criteria that uses a subquery to produce the value set
- `IsNull` – represents an IS NULL criteria
- `Exists` – represents an EXISTS criteria that determines whether a subquery will return any values
- `Like` – represents a LIKE criteria that compares string values

3.4.1.3. The FROM Clause

The FROM clause contains a list of `TableReference`'s.

- `NamedTable` – represents a single Table

- `Join` – has a left and right `TableReference` and information on the join between the items
- `DerivedTable` – represents a table defined by an inline `QueryExpression`

A list of `TableReference` are 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 `Join` s 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 capability "useAnsiJoin" return true. Refer to *Section 3.4.5.3, "Command Form"* for more information.

3.4.1.4. QueryExpression Structure

`QueryExpression` is the base for both SELECT queries and set queries. It may optionally take an `OrderBy` (representing a SQL ORDER BY clause), a `Limit` (represent a SQL LIMIT clause), or a `With` (represents a SQL WITH clause).

3.4.1.5. Select Structure

Each `QueryExpression` can be a `Select` describing the expressions (typically elements) being selected and an `TableReference` specifying the table or tables being selected from, along with any join information. The `Select` may optionally also supply an `Condition` (representing a SQL WHERE clause), a `GroupBy` (representing a SQL GROUP BY clause), an an `Condition` (representing a SQL HAVING clause).

3.4.1.6. SetQuery Structure

A `QueryExpression` can also be a `SetQuery` that represents on of the SQL set operations (UNION, INTERSECT, EXCEPT) on two `QueryExpression`. The all flag may be set to indicate UNION ALL (currently INTERSECT and EXCEPT ALL are not allowed in Teiid)

3.4.1.7. With Structure

A `With` clause contains named `QueryExpressions` held by `WithItem`s that can be referenced as tables in the main `QueryExpression`.

3.4.1.8. Insert Structure

Each `Insert` will have a single `NamedTable` specifying the table being inserted into. It will also has a list of `ColumnReference` specifying the columns of the `NamedTable` that are being inserted into. It also has `InsertValueSource`, which will be a list of Expressions (`ExpressionValueSource`), or a `QueryExpression`, or an Iterator (`IteratorValueSource`)

3.4.1.9. Update Structure

Each `Update` will have a single `NamedTable` specifying the table being updated and list of `SetClause` entries that specify `ColumnReference` and `Expression` pairs for the update. The Update may optionally provide a criteria `Condition` specifying which rows should be updated.

3.4.1.10. Delete Structure

Each `Delete` will have a single `NamedTable` specifying the table being deleted from. It may also optionally have a criteria specifying which rows should be deleted.

3.4.1.11. Call Structure

Each `Call` has zero or more `Argument` objects. The `Argument` objects describe the input parameters, the output result set, and the output parameters.

3.4.1.12. BatchedUpdates Structure

Each `BatchedUpdates` has a list of `Command` objects (which must be either `Insert`, `Update` or `Delete`) that compose the batch.

3.4.2. Language Utilities

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

3.4.2.1. Data Types

The Translator API contains an interface `TypeFacility` that defines data types and provides value translation facilities. This interface can be obtained from calling "getTypeFacility()" method on the "ExecutionFactory" class.

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 `Expression` instances define a data type based on this set of types. These constants are often needed in understanding or creating language interfaces.

3.4.2.2. Language Manipulation

In Translators 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 call "getLanguageFactory()" method on the "ExecutionFactory" to get a reference to the `LanguageFactory` instance for your translator. 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 `Condition` objects are provided in the `LanguageUtil` class. This class has methods to combine `Condition` with AND or to break an `Condition` apart based on AND operators. These utilities are helpful for breaking apart a criteria into individual filters that your translator can implement.

3.4.3. 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. While builing your VDB in the Designer, you can define what called "Extension Model", that defines any number of arbitary properties on a model and its objects. At runtime, using this runtime metadata interface, you get access to those set properties defined during the design time, to define/hint any execution behavior.

Translator gets access to the `RuntimeMetadata` interface at the time of `Excecution` creation. Translators can access runtime metadata by using the interfaces defined in `org.teiid.metadata` package. This package defines API representing a Schema, Table, Columns and Procedures, and ways to navigate these objects.

3.4.3.1. Metadata Objects

All the language objects extend `AbstractMetadataRecord` class

- Column - returns Column metadata record
- Table - returns a Table metadata record
- Procedure - returns a Procedure metadata record
- ProcedureParameter - returns a Procedure Parameter metadata record

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

3.4.3.2. Access to Runtime Metadata

The RuntimeMetadata interface is passed in for the creation of an "Execution". See "createExecution" method on the "ExecutionFactory" class. It provides the ability to look up metadata records based on their fully qualified names in the VDB.

Example 3.1. Obtaining Metadata Properties

The process of getting a Table's properties is sometimes needed for translator development. For example to get the "NameInSource" property or all extension properties:

```
//getting the Table metadata from an Table is straight-forward
Table table = runtimeMetadata.getTable("table-name");
String contextName = table.getNameInSource();

//The props will contain extension properties
```

```
Map<String, String> props = table.getProperties();
```

3.4.4. Language Visitors

3.4.4.1. Framework

The API provides a language visitor framework in the `org.teiid.language.visitor` 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 `AbstractLanguageVisitor` 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 translator 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.

3.4.4.2. 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.

3.4.4.3. 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(ColumnReference)` 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
LanguageObject 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();
```

3.4.5. Translator Capabilities

The `ExecutionFactory` class defines all the methods that describe the capabilities of a Translator. These are used by the Connector Manager to determine what kinds of commands the translator is capable of executing. A base `ExecutionFactory` class implements all the basic capabilities methods, which says your translator does not support any capabilities. Your extended `ExecutionFactory` class must override the the necessary methods to specify which capabilities your translator supports. You should consult the debug log of query planning (set showplan debug) to see if desired pushdown requires additional capabilities.

3.4.5.1. Capability Scope

Note that if your capabilities will remain unchanged for the lifetime of the translator, since the engine will cache them for reuse by all instances of that translator. Capabilities based on connection/user are not supported.

3.4.5.2. Capabilities

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

Table 3.2. Available Capabilities

Capability	Requires	Description
SelectDistinct		Translator can support SELECT DISTINCT in queries.
SelectExpression		Translator can support SELECT of more than just column references.
AliasedTable		Translator can support Tables in the FROM clause that have an alias.
InnerJoins		Translator can support inner and cross joins
SelfJoins	AliasedGroups and at least on of the join type supports.	Translator can support a self join between two aliased versions of the same Table.
OuterJoins		Translator can support LEFT and RIGHT OUTER JOIN.
FullOuterJoins		Translator can support FULL OUTER JOIN.
InlineViews	AliasedTable	Translator can support a named subquery in the FROM clause.
BetweenCriteria		Not currently used - between criteria is rewriten as compound comparisions.
CompareCriteriaEquals		Translator can support comparison criteria with the operator "=".
CompareCriteriaOrdered		Translator can support comparison criteria with the operator ">" or "<".
LikeCriteria		Translator can support LIKE criteria.
LikeCriteriaEscapeCharacter	LikeCriteria	Translator can support LIKE criteria with an ESCAPE character clause.
InCriteria	MaxInCriteria	Translator can support IN predicate criteria.
InCriteriaSubquery		Translator can support IN predicate criteria where values are supplied by a subquery.
IsNullCriteria		Translator can support IS NULL predicate criteria.
OrCriteria		Translator can support the OR logical criteria.
NotCriteria		Translator 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		Translator can support EXISTS predicate criteria.
QuantifiedCompareCriteriaAll		Translator can support a quantified comparison criteria using the ALL quantifier.
QuantifiedCompareCriteriaSome		

Capability	Requires	Description
		Translator can support a quantified comparison criteria using the SOME or ANY quantifier.
OrderBy		Translator can support the ORDER BY clause in queries.
OrderByUnrelated	OrderBy	Translator can support ORDER BY items that are not directly specified in the select clause.
OrderByNullOrdering	OrderBy	Translator can support ORDER BY items with NULLS FIRST/LAST.
GroupBy		Translator can support an explict GROUP BY clause.
Having	GroupBy	Translator can support the HAVING clause.
AggregatesAvg		Translator can support the AVG aggregate function.
AggregatesCount		Translator can support the COUNT aggregate function.
AggregatesCountStar		Translator can support the COUNT(*) aggregate function.
AggregatesDistinct	At least one of the aggregate functions.	Translator can support the keyword DISTINCT inside an aggregate function. This keyword indicates that duplicate values within a group of rows will be ignored.
AggregatesMax		Translator can support the MAX aggregate function.
AggregatesMin		Translator can support the MIN aggregate function.
AggregatesSum		Translator can support the SUM aggregate function.
AggregatesEnhancedNumeric		Translator can support the VAR_SAMP, VAR_POP, STDDEV_SAMP, STDDEV_POP aggregate functions.
ScalarSubqueries		Translator 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.	Translator 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		Translator can support "searched" CASE expressions anywhere that expressions are accepted.
Unions		Translator support UNION and UNION ALL
Intersect		Translator supports INTERSECT
Except		Translator supports Except

Capability	Requires	Description
SetQueryOrderBy	Unions, Intersect, or Except	Translator supports set queries with an ORDER BY
RowLimit		Translator can support the limit portion of the limit clause
RowOffset		Translator 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		Translator supports INSERT statements with values specified by an QueryExpression.
supportsBatchedUpdates		Translator supports a batch of INSERT, UPDATE and DELETE commands to be executed together.
BulkUpdate		Translator supports updates with multiple value sets
InsertWithIterator		Translator supports inserts with an iterator of values. The values would typically be from an evaluated QueryExpression.
CommonTableExpressions		Translator supports the WITH clause.

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

3.4.5.3. Command Form

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

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

3.4.5.4. Scalar Functions

The method `ExecutionFactory.getSupportedFunctions()` can be used to specify which scalar functions the Translator supports. The set of possible functions is based on the set of functions supported by Teiid. This set can be found in the Reference documentation at *http://www.jboss.org/teiid//docs.html*. If the Translator 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.

3.4.5.5. Physical Limits

The method `ExecutionFactory.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 `ExecutionFactory.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.

3.4.5.6. Update Execution Modes

The method `ExecutionFactory.supportsBatchedUpdates()` can be used to indicate that the Translator supports executing the `BatchedUpdates` command.

The method `ExecutionFactory.supportsBulkUpdate()` can be used to indicate that the Translator accepts update commands containg multi valued Literals.

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

3.4.5.7. Default Behavior

The method `ExecutionFactory.getDefaultNullOrder()` specifies the default null order. Can be one of UNKNOWN, LOW, HIGH, FIRST, LAST. This is only used if ORDER BY is supported, but null ordering is not.

3.5. Large Objects

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

3.5.1. 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 translator framework to support memory-safe streaming.

3.5.2. Why Use Large Object Support?

Teiid allows a Translator to return a large object through the Teiid translator 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 Translator 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.

3.5.3. Handling Large Objects

The Translator 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 translators.

A 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. See ExecutionContext.keepAlive() method.

When the "keepAlive" alive flag is set, then the execution object is only closed when user's Statement is closed.

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

3.5.4. Inserting or Updating Large Objects

LOBs will be passed to the Translator in the language objects as Literal 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.

3.6. Packaging

Once the "ExecutionFactory" class is implemented, package it in a JAR file. The only additional requirement is provide a file called "jboss-beans.xml" in the "META-INF" directory of the JAR file, with following contents. Replace ${name} with name of your translator, and replace ${execution-factory-class} with your overridden ExecutionFactory class name. This will register the Translator for use with tooling and Admin API.

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

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

                                        <bean                name="translator-${name}-template"
 class="org.teiid.templates.TranslatorDeploymentTemplate">
    <property name="info"><inject bean="translator-${name}"/></property>
        <property  name="managedObjectFactory"><inject  bean="ManagedObjectFactory"/></
property>
  </bean>

  <bean name="translator-${name}" class="org.teiid.templates.TranslatorTemplateInfo">
    <constructor factoryMethod="createTemplateInfo">
      <factory bean="TranslatorDeploymentTemplateInfoFactory"/>
              <parameter  class="java.lang.Class">org.teiid.templates.TranslatorTemplateInfo</
parameter>
      <parameter class="java.lang.Class">${execution-factory-class}</parameter>
      <parameter class="java.lang.String">translator-${name}</parameter>
      <parameter class="java.lang.String">${name}</parameter>
    </constructor>
  </bean>

</deployment>
```

3.7. Deployment

Copy the JAR file that defines the Translator into "deploy" directory of the JBoss AS's chosen profile, and the Translator will be deployed automatically. There is no restriction that, JBoss AS need to be restarted. However, if your Translator has external dependencies to other JAR libraries, they need to be placed inside the "lib" directory of the JBoss AS's profile. This will require a restart of the JBoss Server. Another option to avoid the restart is to bundle all the required JAR files into the same JAR file as the Translator. It is user's responsibility to make sure they are not running into any conflicts with their dependent libraries with those already exist in the JBoss environment.

Extending The JDBC Translator

The JDBC Translator can be extended to handle new JDBC drivers and database versions. This is one of the most common needs of custom Translator development. This chapter outlines the process by which a user can modify the behavior of the JDBC Translator for a new source, rather than starting from scratch.

To design a JDBC Translator for any RDMS that is not already provided by the Teiid, extend the `org.teiid.translator.jdbc.JDBCExecutionFactory` class in the "translator-jdbc" module. There are three types of methods that you can override from the base class to define the behavior of the Translator.

Table 4.1. Extensions

Extension	Purpose
Capabilities	Specify the SQL syntax and functions the source supports.
SQL Translation	Customize what SQL syntax is used, how source-specific functions are supported, how procedures are executed.
Results Translation	Customize how results are retrieved from JDBC and translated.

4.1. Capabilities Extension

This extension must override the methods that begin with "supports" that describe translator capabilities. Refer to *Section 3.4.5, "Translator Capabilities"* for all the available translator capabilities.

The most common example is adding support for a scalar function – this requires both declaring that the translator has the capability to execute the function and often modifying the SQL Translator to translate the function appropriately for the source.

Another common example is turning off unsupported SQL capabilities (such as outer joins or subqueries) for less sophisticated JDBC sources.

4.2. SQL Translation Extension

The JDBCExcecutionFactory provides several methods to modify the command and the string form of the resulting syntax before it is sent to the JDBC driver, including:

- Change basic SQL syntax options. See the useXXX methods, e.g. useSelectLimit returns true for SQLServer to indicate that limits are applied in the SELECT clause.
- Register one or more FunctionModifiers that define how a scalar function should be modified or transformed.

- Modify a LanguageObject. - see the translate, translateXXX, and FunctionModifiers.translate methods. Modify the passed in object and return null to indicate that the standard syntax output should be used.
- Change the way SQL strings are formed for a LanguageObject. - - see the translate, translateXXX, and FunctionModifiers.translate methods. Return a list of parts, which can contain strings and LanguageObjects, that will be appended in order to the SQL string. If the in coming LanguageObject appears in the returned list it will not be translated again.

Refer to *Section 4.4.1, "Using FunctionModifiers"*.

4.3. Results Translation Extension

The JDBCExecutionFactory provides several methods to modify the java.sql.Statement and java.sql.ResultSet interactions, including:

1. Overriding the createXXXExecution to subclass the corresponding JDBCXXXExecution. The JDBCBaseExecution has protected methods to get the appropriate statement (getStatement, getPreparedStatement, getCallableStatement) and to bind prepared statement values bindPreparedStatementValues.
2. Retrieve values from the JDBC ResultSet or CallableStatement - see the retrieveValue methods.

4.4. Adding Function Support

Refer to *Chapter 5, User Defined Functions* for adding new functions to Teiid. This example will show you how to declare support for the function and modify how the function is passed to the data source.

Following is a summary of all coding steps in supporting a new scalar function:

1. Override the capabilities method to declare support for the function (REQUIRED)
2. Implement a FunctionModifier to change how a function is translated and register it for use (OPTIONAL)

There is a capabilities method getSupportedFunctions() that declares all supported scalar functions.

An example of an extended capabilities class to add support for the "abs" absolute value function:

```
package my.connector;

import java.util.ArrayList;
import java.util.List;
```

```
public class ExtendedJDBCExecutionFactory extends JDBCExecutionFactory
{
  @Override
  public List getSupportedFunctions()
  {
    List supportedFunctions = new ArrayList();
    supportedFunctions.addAll(super.getSupportedFunctions());
    supportedFunctions.add("ABS");
    return supportedFunctions;
  }
}
```

In general, it is a good idea to call super.getSupportedFunctions() to ensure that you retain any function support provided by the translator you are extending.

This may be all that is needed to support a Teiid function if the JDBC data source supports the same syntax as Teiid. The built-in SQL translation will translate most functions as: "function(arg1, arg2, …)".

4.4.1. Using FunctionModifiers

In some cases you may need to translate the function differently or even insert additional function calls above or below the function being translated. The JDBC translator provides an abstract class `FunctionModifier` for this purpose.

During the start method a modifier instance can be registered against a given function name via a call to `JDBCExecutionFactory.registerFunctionModifier`.

The FunctionModifier has a method called `translate`. Use the translate method to change the way the function is represented.

An example of overriding the translate method to change the MOD(a, b) function into an infix operator for Sybase (a % b). The translate method returns a list of strings and language objects that will be assembled by the translator into a final string. The strings will be used as is and the language objects will be further processed by the translator.

```
public class ModFunctionModifier implements FunctionModifier
{
  public List translate(Function function)
  {
    List parts = new ArrayList();
    parts.add("(");
    Expression[] args = function.getParameters();
    parts.add(args[0]);
```

```
    parts.add(" % ");
    parts.add(args[1]);
    parts.add(")");
    return parts;
  }
}
```

In addition to building your own FunctionModifiers, there are a number of pre-built generic function modifiers that are provided with the translator.

Table 4.2. Common Modifiers

Modifier	Description
AliasModifier	Handles simply renaming a function ("ucase" to "upper" for example)
EscapeSyntaxModifier	Wraps a function in the standard JDBC escape syntax for functions: {fn xxxx()}

To register the function modifiers for your supported functions, you must call the `ExecutionFactory.registerFunctionModifier(String name, FunctionModifier modifier)` method.

```
public class ExtendedJDBCExecutionFactory extends JDBCExecutionFactory
{
  @Override
  public void start()
  {
    super.start();

    // register functions.
    registerFunctionModifier("abs", new MyAbsModifier());
    registerFunctionModifier("concat", new AliasModifier("concat2"));
  }
}
```

Support for the two functions being registered ("abs" and "concat") must be declared in the capabilities as well. Functions that do not have modifiers registered will be translated as usual.

4.5. Installing Extensions

Once you have developed an extension to the JDBC translator, you must install it into the Teiid Server. The process of packaging or deploying the extended JDBC translators is exactly as any other other translator. Since the RDMS is accessible already through its JDBC driver, there is

no need to develop a resource adapter for this source as JBoss AS provides a wrapper JCA connector (DataSource) for any JDBC driver.

Refer to *Section 3.6, "Packaging"* and *Section 3.7, "Deployment"* for more details.

User Defined Functions

If you need to extends Teiid's scalar function library, then Teiid provides a means to define custom scalar functions or User Defined Functions(UDF). The following steps need to be taken in creating a UDF.

5.1. UDF Definition

The FunctionDefinition.xmi file provides metadata to the query engine on User Defined Functions. See the Designer Documentation for more on creating a Function Definition Model.

The following are used to define a UDF.

- *Function Name* When you create the function name, keep these requirements in mind:
  - You cannot use a reserved word, which includes existing Teiid System function names. You cannot overload existing Teiid System functions.
  - The function name must be unique among user-defined functions for the number of arguments. You can use the same function name for different numbers of types of arguments. Hence, you can overload your user-defined functions.
  - The function name can only contain letters, numbers, and the underscore (_). Your function name must start with a letter.
  - The function name cannot exceed 128 characters.
- *Input Parameters* - defines a type specific signature list. All arguments are considered required.
- *Return Type* - the expected type of the returned scalar value.
- *Pushdown* - can be one of REQUIRED, NEVER, ALLOWED. Indicates the expected pushdown behavior. If NEVER or ALLOWED are specified then a Java implementation of the function should be supplied.
- *invocationClass/invocationMethod* - optional properties indicating the static method to invoke when the UDF is not pushed down.
- *Deterministic* - if the method will always return the same result for the same input parameters.

Even pushdown required functions need to be added as a UDF to allow Teiid to properly parse and resolve the function. Pushdown scalar functions differ from normal user-defined functions in that no code is provided for evaluation in the engine. An exception will be raised if a pushdown required function cannot be evaluated by the appropriate source.

5.2. Source Supported UDF

While Teiid provides an extensive scalar function library, it contains only those functions that can be evaluated within the query engine. In many circumstances, especially for performance, a user defined function allows for calling a source specific function.

For example, suppose you want to use the Oracle-specific functions score and contains:

```
SELECT score(1), ID, FREEDATA FROM Docs WHERE contains(freedata, 'nick', 1) > 0
```

The `score` and `contains` functions are not part of built-in scalar function library. While you could write your own custom scalar function to mimic their behavior, it's more likely that you would want to use the actual Oracle functions that are provided by Oracle when using the Oracle Free Text functionality.

In addition to the normal steps outlined in the section to create and install a function model (FunctionDefinitions.xmi), you will need to extend the appropriate connector(s).

For example, to extend the Oracle Connector

- *Required* - extend the OracleExecutionFactory and add SCORE and CONTAINS as supported functions. For this example, we'll call the class MyOracleExecutionFactory. Add the `org.teiid.translator.Translator` annotation to the class, e.g. `@Translator(name="myoracle")`
- Optionally register new FunctionModifiers on the start of the ExecutionFactory to handle translation of these functions. Given that the syntax of these functions is same as other typical functions, this probably isn't needed - the default translation should work.
- Create a new translator jar containing your custom ExecutionFactory. Refer to *Section 3.6, "Packaging"* and *Section 3.7, "Deployment"* for instructions on using the JAR file.

5.3. Non-pushdown Support for User-Defined Functions

Non-pushdown support requires a Java function that matches the metadata supplied in the FunctionDefinitions.xmi file. You must create a Java method that contains the function's logic. This Java method should accept the necessary arguments, which the Teiid System will pass to it at runtime, and function should return the calculated or altered value.

5.3.1. Java Code

Code Requirements

- The java class containing the function method must be defined public.
- The function method must be public and static.

- Number of input arguments and types must match the function metadata defined in *Section 5.1, "UDF Definition"*.
- Any exception can be thrown, but Teiid will rethrow the exception as a `FunctionExecutionException` .

You may optionally add an additional `org.teiid.CommandContext` argument as the first parameter. The `CommandContext` interface provides access to information about the current command, such as the executing user, the vdb, the session id, etc. This `CommandContext` parameter does not need to be delared in the function metadata.

Example 5.1. Sample code

```
package org.something;

public class TempConv
{
  /**
  * Converts the given Celsius temperature to Fahrenheit, and returns the
  * value.
  * @param doubleCelsiusTemp
  * @return Fahrenheit
  */
  public static Double celsiusToFahrenheit(Double doubleCelsiusTemp)
  {
    if (doubleCelsiusTemp == null)
    {
      return null;
    }
    return (doubleCelsiusTemp)*9/5 + 32;
  }
}
```

5.3.2. Post Code Activities

1. After coding the functions you should compile the Java code into a Java Archive (JAR) file.
2. The JAR should be available in the classpath of Teiid - this could be the server profile lib, or the deployers/teiid.deployer directory depending upon your preference.

5.4. Installing user-defined functions

Once a user-defined function model (FunctionDefinitions.xmi) has been created in in the Designer Tool, it can be added to the VDB for use by Teiid.

AdminAPI

In most circumstances the admin operations will be performed through the admin console or AdminShell tooling, but it is also possible to invoke admin functionality directly in Java through the AdminAPI.

All classes for the AdminAPI are in the client jar under the `org.teiid.adminapi` package.

6.1. Connecting

An AdminAPI connection, which is represented by the `org.teiid.adminapi.Admin` interface, is obtained through the `org.teiid.adminapi.AdminFactory.createAdmin` methods. `AdminFactory` is a singleton, see `AdminFactory.getInstance()`. The `Admin` instance automatically tests its connection and reconnects to a server in the event of a failure. The `close` method should be called to terminate the connection.

See your Teiid installation for the appropriate admin port - the default is 31443.

6.2. Admin Methods

Admin methods exist for monitoring, server administration, and configuration purposes. Note that the objects returned by the monitoring methods, such as getRequests, are read-only and cannot be used to change server state. See the JavaDocs for all of the details.

Logging

7.1. Customized Logging

The Teiid system provides a wealth of information using logging. To control logging level, contexts, and log locations, you should be familiar with log4j and the container's `jboss-log4j.xml` configuration file. Teiid also provides a *`PROFILE`*`/conf/jboss-teiid-log4j.xml` containing much of information from this chapter. Refer to the Administrator Guide for more details about different Teiid contexts available. Refer to *http://logging.apache.org/log4j/* for more information about log4j.

If the default log4j logging mechanisms are not sufficient for your logging needs you may need a different appender, refer to the log4j javadocs at *http://logging.apache.org/log4j/1.2/apidocs/index.html*. Note that log4j already provides quite a few appenders including JMS, RDBMS, and SMTP.

If you want a custom appender, follow the Log4J directions to write a custom appender. Refer to the instructions at *http://logging.apache.org/log4net/release/faq.html*. If you develop a custom logging solution, the implementation jar should be placed in the "lib" directory of the JBoss AS server profile Teiid is installed in.

7.1.1. Command Logging API

If you want to build a custom appender for command logging that will have access to log4j "LoggingEvents" to the "COMMAND_LOG" context, it will have a message that is an instance of `org.teiid.logging.CommandLogMessage` defined in the `teiid-api-7.2.jar` use these class in your development. The CommmdLogMessage include information about vdb, session, command-sql etc.

7.1.2. Audit Logging API

If you want to build a custom appender for command logging that will have access to log4j "LoggingEvents" to the "AUDIT_LOG" context, it will have a message that is an instance of `org.teiid.logging.AuditMessage` defined in the `teiid-api-7.2.jar` use this class in your development. AuditMessage include information about user, the action, and the target(s) of the action.

Login Modules

The Teiid system provides a range of built-in and extensible security features to enable the secure access of data. For details about how to configure the available security features check out Admin Guide.

LoginModules are an essential part of the JAAS security framework and provide Teiid customizable user authentication and the ability to reuse existing LoginModules defined for JBossAS. Refer to the JBoss Application Server security documentation for information about configuring security in JBoss Application Server, *http://docs.jboss.org/jbossas/admindevel326/html/ch8.chapter.html*.

8.1. Built-in LoginModules

JBoss Application Server provides several LoginModules for common authentication needs, such as authenticating from text files or LDAP.

Below are are some of those available in JBoss Application Server:

UserRoles LoginModule
: Login module that uses simple file based authentication.

  Refer to *http://community.jboss.org/docs/DOC-12510*.

LDAP LoginModule
: Login module that uses LDAP based authentication.

  Refer to *http://community.jboss.org/docs/DOC-11253*.

Database LoginModule
: Login module that uses Database-based authentication.

  Refer to *http://community.jboss.org/docs/DOC-9511*.

Cert LoginModule
: Login module that uses X509 certificate based authentication.

  See *http://community.jboss.org/docs/DOC-9160*.

For all the available login modules refer to *http://community.jboss.org/docs/DOC-11287*.

8.2. Custom LoginModules

If your authentication needs go beyond the provided LoginModules, please refer to the JAAS development guide at *http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/JAASLMDevGuide.html*. There are also numerous guides available.

If you are extending one of the built-in LoginModules, refer to *http://community.jboss.org/docs/DOC-9466*.

Appendix A. ra.xml file Template

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

```
<?xml version="1.0" encoding="UTF-8"?>
<connector xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd" version="1.5">

  <vendor-name>${comapany-name}</vendor-name>
  <eis-type>${type-of-connector}</eis-type>
  <resourceadapter-version>1.0</resourceadapter-version>
  <license>
    <description>${license text}</description>
    <license-required>true</license-required>
  </license>

  <resourceadapter>
        <resourceadapter-class>org.teiid.resource.spi.BasicResourceAdapter</resourceadapter-
class>
    <outbound-resourceadapter>
      <connection-definition>
          <managedconnectionfactory-class>${connection-factory}</managedconnectionfactory-
class>

        <!-- repeat for every configuration property -->
        <config-property>
          <description>
            {$display:"${short-name}",$description:"${description}",$allowed:[${value-list}],
            $required:"${required-boolean}", $defaultValue:"${default-value}"}
          </description>
          <config-property-name>${property-name}</config-property-name>
          <config-property-type>${property-type}</config-property-type>
          <config-property-value>${optional-property-value}</config-property-value>
        </config-property>

        <!-- use the below as is if you used the Connection Factory interface -->
        <connectionfactory-interface>
          javax.resource.cci.ConnectionFactory
        </connectionfactory-interface>
```

```
        <connectionfactory-impl-class>
          org.teiid.resource.spi.WrappedConnectionFactory
        </connectionfactory-impl-class>

        <connection-interface>
          javax.resource.cci.Connection
        </connection-interface>

        <connection-impl-class>
          org.teiid.resource.spi.WrappedConnection
        </connection-impl-class>

      </connection-definition>

      <transaction-support>NoTransaction</transaction-support>

      <authentication-mechanism>
        <authentication-mechanism-type>BasicPassword</authentication-mechanism-type>
        <credential-interface>
          javax.resource.spi.security.PasswordCredential
        </credential-interface>
      </authentication-mechanism>
      <reauthentication-support>false</reauthentication-support>

    </outbound-resourceadapter>

  </resourceadapter>

</connector>
```

${...} indicates a value to be supplied by the developer.

Appendix B. Advanced Topics

B.1. Security Migration From Previous Versions

It is recommended that customers who have utilized the internal JDBC membership domain from releases prior to MetaMatrix 5.5 migrate those users and groups to an LDAP compliant directory server.

Refer to the JBoss Application Server security documentation for using an LDAP directory server. If there are additional questions or the need for guidance in the migration process, please contact technical support.

Several free and open source directory servers include:

- The Fedora Directory Server - *http://directory.fedoraproject.org/*
- Open LDAP - *http://www.openldap.org/*
- Apache Directory Server - *http://directory.apache.org/*