Red Hat JBoss Data Virtualization 6.2

Development Guide Volume 4: Server Development

This guide is intended for developers

Last Updated: 2017-10-20

Red Hat JBoss Data Virtualization 6.2 Development Guide Volume 4: Server Development

This guide is intended for developers

Red Hat Customer Content Services

Legal Notice

Copyright © 2017 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document provides information for developers creating custom solutions.

Table of Contents

CHAPTER 1. READ ME

1.1. BACK UP YOUR DATA

WARNING

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

1.2. VARIABLE NAME: EAP_HOME

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

1.3. VARIABLE NAME: MODE

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

1.4. RED HAT DOCUMENTATION SITE

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

CHAPTER 2. EMBEDDED JBOSS DATA VIRTUALIZATION (TECHNOLOGY PREVIEW)

2.1. TECHNOLOGY PREVIEW

WARNING

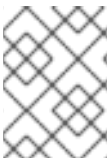
Technology preview features provide early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. However, these features are not fully supported under Subscription Level Agreements, may not be functionally complete, and are not intended for production use. As Red Hat considers making future iterations of technology preview features generally available, we will attempt to resolve any issues that customers experience when using these features. During the development of a technology preview feature, additional components may become available to the public for testing. Because technology preview features are still under development, Red Hat cannot guarantee the stability of such features. As a result, if you are using technology preview features, you may not be able to seamlessly upgrade to subsequent releases of that feature. While Red Hat intends to fully support technology preview features in future releases, we may discover that a feature does not meet the standards for enterprise viability. If this happens, we cannot guarantee that technology preview features will be released in a supported manner. Some technology preview features may only be available for specific hardware architectures.

NOTE

Red Hat JBoss support will provide commercially reasonable efforts to resolve any reported issues that customers experience when using these features.

2.2. EMBEDDED JBOSS DATA VIRTUALIZATION

Embedded JBoss Data Virtualization is a lightweight version of JBoss Data Virtualization for use in any Java 6+ JRE. Red Hat JBoss Enterprise Application Platform (or any other application server) is not required.



NOTE

This feature/kit is still evolving. Consult the source examples and unit tests utilizing the `EmbeddedServer` for more guidance.

2.3. CONFIGURATION

The primary way to configure Embedded JBoss Data Virtualization is with the `EmbeddedConfiguration` class. It is provided to the `EmbeddedServer` at start-up. From there the running server instance can have translators and VDBs deployed as needed.

Your application is responsible for having the appropriate classpath to utilize Teiid embedded. Typically you will want to include all of the jars from the embedded kit's lib directory. As needed by your deployment you should include jars from the optional folder along with any jars needed to provide source access. Hibernate core 4.1.6 or compatible is needed, but not included in the kit, if you wish to utilize the JDBC translator support for dependent joins using temporary tables.

NOTE

All Teiid jars can also be deployed as bundles in a OSGI container like Karaf. If you are working with Karaf, a feature.xml file is available in maven repo for your convenience. Here is the usage pattern:

```
features:addurl
mvn:org.jboss.teiid/teiid/8.6.0.Final/xml/karaf-features
features:install -v teiid
```

If you are trying run Teidd Embedded with a Maven-based project and you are using Maven to pull artifacts, the runtime, admin, connector, translator dependencies are required:

```
<dependency>
    <groupId>org.jboss.teiid</groupId>
    <artifactId>teiid-runtime</artifactId>
</dependency>

<dependency>
    <groupId>org.jboss.teiid</groupId>
    <artifactId>teiid-admin</artifactId>
</dependency>

<dependency>
    <groupId>org.jboss.teiid.connectors</groupId>
    <artifactId>translator-SOURCE</artifactId>
</dependency>

<dependency>
   <groupId>org.jboss.teiid.connectors</groupId>
   <artifactId>connector-SOURCE</artifactId>
   <classifier>lib</classifier>
</dependency>
```

2.4. VDB DEPLOYMENT

VDB deployment can be done directly through VDB metadata objects that are the underpinning of vdb.xml deployment. Models (schemas) are deployed as a set to form a named VDB (see the **`EmbeddedServer.deployVDB`** method).

Typically there is no concept of VDB versioning.

XML Deployment

Similar to a server based -vdb.xml deployment, an **`InputStream`** may be given to a vdb.xml file. See the **`EmbeddedServer.deployVDB(InputStream)`** method.

ZIP Deployment

Similar to a server based .vdb deployment, a **URL** may be given to a ZIP file. See the **`EmbeddedServer.deployVDBZip`** method. The use of the ZIP lib for dependency loading is not enabled in the embedded version.

2.5. TRANSLATORS

When running JBoss Data Virtualization with JBoss EAP, translator instances are scoped to a VDB using declarations in a **`vdb.xml`** file; however, for Embedded JBoss Data Virtualization, instances of translators are scoped to the entire **`EmbeddedServer`** and must be registered via the **`EmbeddedServer.addTranslator`** method. A new server instance does not assume any translators are deployed and does not perform any library scanning to find translators.

2.6. SOURCES

The **`EmbeddedServer`** will still attempt to lookup the given JNDI connection factory names via JNDI. In most non-container environments it is likely that no such bindings exist. In this case the embedded server instance must have **`ConnectionFactoryProvider`** instances manually registered using the **`EmbeddedServer.addConnectionFactoryProvider`** method. Note that the embedded server does not have built-in pooling logic, so to make better use of a standard **`javax.sql.DataSource`** or to enable proper use of **`javax.sql.XADataSource`** you must first configure the instance via a third-party connection pool.

2.7. EXAMPLE DEPLOYMENT

```
EmbeddedServer es = new EmbeddedServer();
EmbeddedConfiguration ec = new EmbeddedConfiguration();
//set any configuration properties
ec.setUseDisk(false);
es.start(ec);
//example of adding a translator by class - this will make a default
instance available with the default name of oracle
es.addTranslator(OracleExecutionFactory.class);

//add a translator by instance - this is functionally equivalent to using
a vdb.xml translator override
OracleExecutionFactory oef = new OracleExecutionFactory();
//configure and start the instance
oef.setDatabaseVersion("11.0");
oef.start();
es.addTranslator("my-oracle", oef);

//add a connection factory provider if needed
//the default is to perform a jndi lookup of the datasource names given
//however out of a container you will likely need to manually inject the
necessary connection factory

ConnectionFactoryProvider<DataSource> cfp = new
EmbeddedServer.SimpleConnectionFactoryProvider<DataSource>(...);
es.addConnectionFactoryProvider("ora-ds", cfp);

//add a vdb
```

```
//physical model
ModelMetaData mmd = new ModelMetaData();
mmd.setName("my-schema");
mmd.addSourceMapping("my-schema", "my-oracle", "ora-ds");

//virtual model
ModelMetaData mmd1 = new ModelMetaData();
mmd1.setName("virt");
mmd1.setModelType(Type.VIRTUAL);
mmd1.setSchemaSourceType("ddl");
mmd1.setSchemaText("create view \"my-view\" OPTIONS (UPDATABLE 'true') as
select * from \"my-table\"");

es.deployVDB("test", mmd, mmd1);
```

2.8. TRANSACTIONS

For transaction processing, the **`TransactionManager`** used to start the **`EmbeddedServer`** must be set in the **`EmbeddedConfiguration`**. A client facing **`javax.sql.DataSource`** is not provided for Embedded JBoss Data Virtualization. However, the use of **`java.sql.Driver`** should be sufficient as, by default, the embedded server can detect thread bound transactions and appropriately propagate the transaction to threads launched as part of request processing. Use of local connections is also permitted.

2.9. OTHER DIFFERENCES BETWEEN THE EMBEDDED AND EAP DEPLOYMENTS

- There is no JDBC/ODBC socket transport in Embedded JBoss Data Virtualization. You are expected to obtain a **`Driver`** connection via the **`EmbeddedServer.getDriver`** method.
- When running JBoss Data Virtualization with JBoss EAP, a **`MetadataRepository`** is scoped to a VDB, but for Embedded JBoss Data Virtualization is scoped to the entire **`EmbeddedServer`** instance and must be registered via the **`EmbeddedServer.addMetadataRepository`** method.
- MDC logging values are not available as Java logging lacks the concept of a mapped diagnostic context.

CHAPTER 3. DEVELOPING FOR JBOSS DATA VIRTUALIZATION

3.1. DEVELOPING FOR JBOSS DATA VIRTUALIZATION

JBoss Data Virtualization provides several *translators* and *resource adapters* to enable communication with various datasources.

If none of the included translators and resource adapters meet your requirements, you can extend them or create your own. One of the most common examples of custom translator development is the extension of the JDBC translator for new JDBC drivers and database versions.

3.2. JBOSS DATA VIRTUALIZATION CONNECTOR ARCHITECTURE

The process of integrating data from an enterprise information system into JBoss Data Virtualization requires one to two components:

1. a translator (mandatory) and
2. a resource adapter (optional), also known as a connector. Most of the time, this will be a Java EE Connector Architecture (JCA) Adapter.

A translator is used to:

- translate JBoss Data Virtualization commands into commands understood by the datasource for which the translator is being used,
- execute those commands,
- return batches of results from the datasource, translated into the formats that JBoss Data Virtualization is expecting.

A resource adapter (or connector):

- handles all communications with individual enterprise information systems, (which can include databases, data feeds, flat files and so forth),
- can be a JCA Adapter or any other custom connection provider (the JCA specification ensures the writing, packaging and configuration are undertaken in a consistent manner),

NOTE

Many software vendors provide JCA Adapters to access different systems. Red Hat recommends using vendor-supplied JCA Adapters when using JMS with JCA. See http://docs.oracle.com/cd/E21764_01/integration.1111/e10231/adptr_jms.htm

- removes concerns such as connection information, resource pooling, and authentication for translators.

With a suitable translator (and optional resource adapter), any datasource or Enterprise Information System can be integrated with JBoss Data Virtualization.

3.3. TRANSLATORS IN JBOSS DATA VIRTUALIZATION

JBoss Data Virtualization provides the following translators:

Apache Cassandra (Technical Preview Only)

WARNING

Technology Preview features are not supported, may not be functionally complete, and are not intended for production use. These features are included to provide customers with early access to upcoming product innovations, enabling them to test functionality and provide feedback during the development process.

Support of Apache Cassandra brings support for the popular columnar NoSQL database to JDV customers.

Apache Solr

With Apache Solr, JDV customers will be able to take advantage of enterprise search capabilities for organized retrieval of structured and unstructured data.

Cloudera Impala

Cloudera Impala support provides for fast SQL query access to data stored in Hadoop.

JDBC Translator

The JDBC Translator works with many relational databases.

JBoss Enterprise Data Services Platform Supported Configurations

File Translator

The File Translator provides a procedural way to access the file system in order to handle text files.

Google Spreadsheet Translator

The Google Spreadsheet Translator is used to connect to a Google Spreadsheet.

JBoss Data Grid 6.3

You can perform reads and writes to JDG. You can use it as an embedded cache or a remote cache.

LDAP Translator

The LDAP Translator provides access to LDAP directory services.

MongoDB Translator

The MongoDB translator, known by the type name *mongodb*, provides a relational view of data that resides in a MongoDB database. This translator is capable of converting JBoss Data Virtualization SQL queries into MongoDB based queries. It supports a full range of SELECT, INSERT, UPDATE and DELETE calls.

Object Translator

The Object translator is a bridge for reading Java objects from external sources such as JBoss Data Grid (`infinispan-cache`) or Map Cache and delivering them to the engine for processing.

OData Translator

The OData translator exposes the OData V2 and V3 data sources and uses the JBoss Data Virtualization WS resource adapter for making web service calls. This translator is an extension of the WS Translator.

OLAP Translator

The OLAP Services translator exposes stored procedures for calling analysis services backed by an OLAP server using MDX query language.

Salesforce Translator

The Salesforce Translator works with Salesforce interfaces.

Web Services Translator

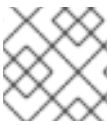
The Web Services Translator provides procedural access to XML content by using *web services*.

If these translators are not suitable for your system then you can develop a custom one.

3.4. RESOURCE ADAPTERS IN JBOSS DATA VIRTUALIZATION

With the exception of JDBC data sources, JBoss Data Virtualization provides a JCA adapter for each supported data source. These are the resource adapter identifiers, as specified in the server configuration file:

- File Adapter - `file`
- Google Spreadsheet Adapter - `google`
- Red Hat JBoss Data Grid (6.1 & 6.2) Adapter - `infinispan`
- LDAP Adapter - `ldap`
- Salesforce Adapter - `salesforce`
- Web Services Adapter - `webservice`
- Mongo DB Adapter (technical preview) - `mongodb`



NOTE

A resource adapter for the JDBC translator is provided with JBoss EAP by default.

3.5. OTHER JBOSS DATA VIRTUALIZATION DEVELOPMENT

JBoss Data Virtualization is highly extensible in other ways:

- You can add *user defined functions*. See Section 12.1, “User Defined Functions”.

- You can adapt logging to your requirements, which is especially useful for custom audit or command logging. See Section 14.1, “Customized Logging”.
- A *delegating translator* can be used to add custom code to all methods for a given translator. See Section 9.3, “Delegating Translator”.
- You can also customize authentication and authorization modules. See the *Red Hat JBoss Data Virtualization Security Guide*.

3.6. SETTING THE DEVELOPMENT ENVIRONMENT

For JBoss Developer Studio, create an empty java project and add "teiid-common-core", "teiid-api" and JEE "connector-api" JARs as dependencies.

For using Maven, use the following dependencies:

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

The **`teiid-version`** property must be set to the expected version. You must also add the new declared property connector-api-version. You can find relevant artifacts in the Maven repository.

3.7. MAVEN REPOSITORY LOCATION

The URL of the repository will depend on where the repository is located on the filesystem, or web server.

File System

- JBoss Data Virtualization - **`file:///path/to/repo/jboss-dv-6.2.0-maven-repository`**

Apache Web Server

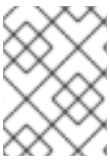
`http://maven.repository.redhat.com/techpreview/all/`

CHAPTER 4. RESOURCE ADAPTER DEVELOPMENT

4.1. DEVELOPING CUSTOM ADAPTERS

For situations in which an existing JCA Adapter (or other connector mechanism) is not suitable, JBoss Data Virtualization provides a framework for developing custom JCA Adapters.

JBoss Data Virtualization uses standard JCA Adapters. Base classes for all of the required supporting JCA SPI (Service Provider Interface) classes are provided by the JBoss Data Virtualization API. The JCA CCI (Common Client Interface) support is not provided because JBoss Data Virtualization uses the translator API as its common client interface.

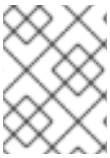


NOTE

If you are not familiar with the JCA API, read the JCA 1.5 Specification at http://docs.oracle.com/cd/E15523_01/integration.1111/e10231/intro.htm.

The process for developing a JBoss Data Virtualization JCA Adapter is as follows (the required classes can be found in org.teiid.resource.spi):

- Define a Managed Connection Factory by extending the **`BasicManagedConnectionFactory`** class
- Define a Connection Factory by extending the **`BasicConnectionFactory`** class
- Define a Connection by extending the **`BasicConnection`** class
- Specify configuration properties in an **`ra.xml`** file



NOTE

The examples contained in this book are simplified and do not include support for transactions or security which would add significant complexity.

For sample resource adapter code, see the **`teiid/connectors`** directory of the **`JBoss Data Virtualization VERSION Source Code`** ZIP file. This ZIP file can be downloaded from the Red Hat Customer Portal at https://access.redhat.com.

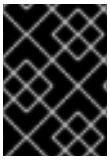
4.2. DEFINE A MANAGED CONNECTION FACTORY

- Extend the **`org.teiid.resource.spi.BasicManagedConnectionFactory`** class, providing an implementation for the **`createConnectionFactory()`** method. This method will create and return an instance of a Connection Factory.
- Define an attribute for each configuration variable, and then provide both "getter" and "setter" methods for them. This class will define various configuration variables (such as user, password, and URL) used to connect to the datasource.

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; }

}
```



IMPORTANT

Use only **`java.lang`** objects as the attributes. DO NOT use Java primitives for defining and accessing the properties.

4.3. DEFINE A CONNECTION FACTORY

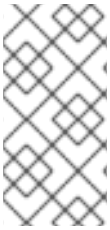
Extend the **`org.teiid.resourse.spi.BasicConnectionFactory`** class, and provide an implementation for the **`getConnection()`** method. This method will create and return an instance of a connection.

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

Since the Managed Connection Factory creates a Connection Factory, it has access to all the configuration parameters so that the **`getConnection()`** method can pass credentials to the requesting application. Therefore, the Connection Factory can reference the calling user's **`javax.security.auth.Subject`** from within the **`getConnection()`** method.

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

A **`Subject`** can give access to logged-in user's credentials and roles that are defined. This may be **`null`**.



NOTE

You can define a *security-domain* for this resource adapter that is separate from the default JBoss Data Virtualization security-domain for validating the JDBC user. However, it is the user's responsibility to perform the necessary logins before the application server's thread accesses this resource adapter.

4.4. DEFINE A CONNECTION

Extend the **`org.teiid.resource.spi.BasicConnection`** class, and provide an implementation based on your access of the Connection object in your translator. If your connection is stateful, override the **`isAlive()`** and **`cleanup()`** methods with suitable implementations. These methods are called to check if a connection is stale and needs flushing from the connection pool by the application server.

```
public class MyConnection extends BasicConnection
{
   public void doSomeOperation(command)
   {
      // do some operation with requesting application..
      // 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()
   {

   }
}
```

4.5. XA TRANSACTIONS

If the requesting application can participate in XA transactions, then your **`Connection`** object must override the **`getXAResource()`** method and provide the **`XAResource`** object for the application. To participate in crash recovery you must also extend the **`BasicResourceAdapter`** class and implement the **`public XAResource[] getXAResources(ActivationSpec[] specs)`** method.

JBoss Data Virtualization can make XA-capable resource adapters participate in distributed transactions. If they are not XA-capable, the datasource can participate in distributed queries but not distributed transactions. Transaction semantics are determined by how you configured "connection-factory" in a "resource-adapter" (that is, jta=true/false).

4.6. SPECIFY CONFIGURATION PROPERTIES IN AN RA.XML FILE

Every configuration property defined inside the new Managed Connection Factory class must also be configured in the **`ra.xml`** file. These properties are used to configure each instance of the connector.

The **ra.xml** file is located in **EAP_HOME/modules/system/layers/dv/org/jboss/teiid/resource-adapter/ADAPTER-NAME/main/META-INF**. An example file is provided in Section C.1, "Template for ra.xml".

The following is the format for a single entry:

```
<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>optional-property-value</config-property-value>
</config-property>
```

For example:

```
<config-property>
   <description>
      {$display:"User Name",$description:"The name of the user.",
$required="true"}
   </description>
   <config-property-name>UserName</config-property-name>
   <config-property-type>java.lang.String</config-property-type>
</config-property>
```

The format and contents of the **<description>** element may be used as extended metadata for tooling. This use of the special format and all properties is optional and must follow these rules:

- The special format must begin and end with curly braces e.g. **{ }**.
- Property names begin with **$**.
- Property names and the associated value are separated with a colon (**:**).
- Double quotes (**"**) identifies a single value.
- A pair of square brackets (**[ ]**), containing comma separated double quoted entries indicates a list value.

The following are optional 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.

- **$advanced**: Notes this as Advanced property.
- **$readOnly**: Property is set to read-only.

NOTE

Although these are optional properties, in the absence of this metadata, JBoss Data Virtualization tooling may not work as expected.

4.7. PACKAGING THE ADAPTER

When development is complete, the resource adapter files are packaged into a deployable artifact called a Resource Adapter Archive or RAR file.

NOTE

The file format is defined by the JCA specification and must not be confused with the RAR file compression format.

The method of creating a RAR artifact will depend on your build system:

JBoss Developer Studio

If you create a Java Connector project in JBoss Developer Studio, it will include a build target that produces a RAR file.

Apache Ant

When using Apache Ant, you can use the standard **rar** build task.

Apache Maven

When using Apache Maven, set the value of the **<packaging>** element to **rar**. Since JBoss Data Virtualization uses Maven, you can refer to any of the Connector projects; for example, **pom.xml** shown below.

```
<project>
   <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>
```

The RAR file under its associated **META-INF** directory must contain the **ra.xml** file. If you are using Apache Maven, see 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.

4.8. ADDING DEPENDENT LIBRARIES

Add a **MANIFEST.MF** file into the **META-INF** directory, and the following line to add the core JBoss Data Virtualization API dependencies for the resource adapter.

```
Dependencies: org.jboss.teiid.common-core,org.jboss.teiid.api,javax.api
```

If your resource adapter depends on any other third party .jar, .dll, or .so files they can be placed at the root of the RAR file. If any of these libraries are already available as modules in JBoss EAP, then you can add the module name to the above **MANIFEST.MF** file to specify them as dependencies.

4.9. DEPLOYING THE ADAPTER

Once the RAR file is built, deploy it using the CLI or Management Console.

Once the adapter's RAR file has been deployed you can create an instance of this connector to use with your Translator. Creating an instance of this adapter is the same as creating a Connection Factory. There are two ways you can do this:

1. Edit the server configuration file and add the following XML in the "resource-adapters" subsystem.

```
<!-- If susbsytem is already defined, only copy the contents under
it and edit to suit your needs -->
<subsystem xmlns="urn:jboss:domain:resource-adapters:1.0">
    <resource-adapters>
        <resource-adapter>
            <archive>teiid-connector-sample.rar</archive>
            <transaction-support>NoTransaction</transaction-
support>
            <connection-definitions>
                <connection-definition class-
name="org.teiid.resource.adapter.MyManagedConnectionFactory"  jndi-
name="${jndi-name}"
                      enabled="true"
                      use-java-context="true"
                      pool-name="sample-ds">
                    <config-property
name="UserName">jdoe</config-property>
                    <config-property name="Count">12</config-
property>
```

```
                </connection-definition>
            </connection-definitions>
        </resource-adapter>
    </resource-adapters>
</subsystem>
```

There are more properties that you can define in this file; for example, for pooling, transactions, and security. Refer to the Red Hat JBoss Enterprise Application Platform documentation for all the available properties. See https://access.redhat.com/site/documentation/JBoss_Enterprise_Application_Platform/.

2. You can use the web-based Management Console to create a new **`ConnectionFactory`**.

CHAPTER 5. TRANSLATOR DEVELOPMENT

5.1. DEVELOPING CUSTOM TRANSLATORS

To create a new custom translator:

1. Create a new (or reuse an existing) resource adapter for the datasource, to be used with this translator.
2. Decide whether to use the Teiid archetype template to create your initial custom translator project and classes or manually create your environment.
3. Create an **`ExecutionFactory`** by:
   - extending the **`org.teiid.translator.ExecutionFactory`** class or
   - extending the **`org.teiid.translator.jdbc.JDBCExecutionFactory`** class .
4. Package the translator.
5. Deploy your translator.
6. Deploy a Virtual Database (VDB) that uses your translator.
7. Execute queries via the Teiid engine.

For sample translator code, refer to the **`teiid/connectors`** directory of the **`JBoss Data Virtualization *VERSION* Source Code`** ZIP file which can be downloaded from the Red Hat Customer Portal at https://access.redhat.com.

To set up the environment for developing a custom translator, you can either manually configure the build environment, structure and framework classes and resources or use the Teiid Translator Archetype template to generate the initial project.

To create the build environment in Red Hat JBoss Developer Studio without any Maven integration, create a Java project and add dependencies to "teiid-common-core", "teiid-api" and JEE "connector-api" jars. However, if you wish to use Maven, add these dependencies:

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

```
    </dependency>
</dependencies>
```

In this case, the ${teiid-version} property should be set to the expected version, such as 8.9.0.Final. You can find Teiid artifacts in the JBoss maven repository .

One way to start developing a custom translator is to create a project using the Teiid archetype template. When the project is created from the template, it will contain the essential classes (in other words, the ExecutionFactory) and resources for you to begin adding your custom logic. Additionally, the maven dependencies are defined in the pom.xml file so that you can begin compiling the classes.

The first way to create a translator project is by using JBoss Developer Studio:

Procedure 5.1. Create a Project in JBDS

1. Open the Java perspective
2. From the menu select File - New - Other.
3. In the tree, expand Maven and select Maven Project.
4. Click Next.
5. On the "Select project name and Location" window, you can accept the defaults, so click Next
6. On the "Select an Archetype" window, click the Configure button
7. Add the remote catalog found at https://repository.jboss.org/nexus/content/repositories/releases/ then click OK to return.
8. Enter "teiid" in the filter to see the Teiid archetypes.
9. Select the translator-archetype 8.7.x and then click Next.
10. Enter all the information (such as Group ID and, Artifact ID) needed to generate the project.
11. Click Finish.

The other method involves using the command line.

Procedure 5.2. Create a Project Using the Command Line

1. Issue the following template command: **`mvn archetype:generate \ -DarchetypeGroupId=org.jboss.teiid.arche-types \ -DarchetypeArtifactId=translator-archetype \ -DarchetypeVersion=8.7.0 \ -DgroupId=${groupId} \ -DartifactId=translator-${translator-name} \ -Dpackage=org.teiid.translator.${translator-name} \ -Dversion=${version} \ -Dtranslator-name=${translator-name} \ -Dteiid-version=${teiid-version}`**

   This is what the instructions mean:

   - -DarchetypeGroupId - is the group ID for the archetype to use to generate
   - -DarchetypeArtifactId - is the artifact ID for the archetype to use to generate

- -DarchetypeVersion - is the version for the archetype to use to generate
- -DgroupId - (user defined) group ID for the new translator project pom.xml
- -DartifactId - (user defined) artifact ID for the new translator project pom.xml
- -Dpackage - (user defined) the package structure where the java and resource files will be created
- -Dversion - (user defined) the version that the new connector project pom.xml will be
- -Dtranslator-name - (user defined) the name (type) of the new translator project, used to create the java class names
- -Dteiid-version - the Teiid version upon which the connector will depend.

Here is a sample command: **`mvn archetype:generate \ -DarchetypeGroupId=org.jboss.teiid.arche-types \ -DarchetypeArtifactId=translator-archetype \ -DarchetypeVersion=8.7.0 \ -DgroupId=org.jboss.teiid.connector \ -DartifactId=translator-myType \ -Dpackage=org.teiid.translator.myType \ -Dversion=0.0.1-SNAPSHOT \ -Dtranslator-name=MyType \ -Dteiid-version=8.7.0.Final`**

2. After you execute it, you will be asked to confirm the properties:

```
Confirm properties configuration:
groupId: org.jboss.teiid.connector
artifactId: translator-myType
version: 0.0.1-SNAPSHOT
package: org.teiid.translator.myType
teiid-version: 8.7.0.Final
translator-name: MyType
 Y: :
```

Type Y (for Yes) and click enter.

3. Upon creation, a directory based on the artifactId will be created, that will contain the project. Navigate to that directory.

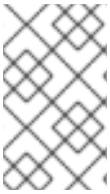
4. Execute a test build to confirm the project was created correctly: **`mvn clean install`**

It should build successfully. If so, you are now ready to start adding your custom code.

5.2. IMPLEMENTING THE FRAMEWORK

Translators may contribute cache entries to the result set cache by the use of the CacheDirective object. Translators wishing to participate in caching should return a CacheDirective from the **`ExecutionFactory.getCacheDirective`** method, which is called prior to execution. The command passed to getCacheDirective will already have been vetted to ensure that the results are eligible for caching. For example update commands or commands with pushed dependent sets will not be eligible for caching.

If the translator returns null for the CacheDirective, which is the default implementation, the engine will not cache the translator results beyond the current command. It is up to your custom translator or custom delegating translator to implement your desired caching policy.



NOTE

In special circumstances where the translator has performed its own caching, it can indicate to the engine that the results should not be cached or reused by setting the Scope to Scope.NONE.

The returned CacheDirective will be set on the ExecutionContext and is available via the **`ExecutionContext.getCacheDirective()`** method. Having ExeuctionFactory.getCacheDirective called prior to execution allows the translator to potentially be selective about which results to even attempt to cache. Since there is a resource overhead with creating and storing the cached results it may not be desirable to attempt to cache all results if it is possible to return large results that have a low usage factor. If you are unsure about whether to cache a particular command result you may return an initial CacheDirective then change the Scope to Scope.NONE at any time prior to the final cache entry being created and the engine will give up creating the entry and release its resources.

NOTE

If you plan on modifying the CacheDirective during execution, ensure that you return a new instance from the ExecutionFactory.getCacheDirective call, rather than returning a shared instance.

The CacheDirective readAll Boolean field is used to control whether the entire result should be read if not all of the results were consumed by the engine. If readAll is false then any partial usage of the result will not result in it being added as a cache entry. Partial use is determined after any implicit or explicit limit has been applied. The other fields on the CacheDirective object map to the cache hint options.

Table 5.1. Options

Option	Default
scope	Session
ttl	rs cache ttl
readAll	true
updatable	true
prefersMemory	false

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.

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

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.
- Parameter – represents a parameter with multiple values. The command should be an instance of BatchedCommand, which provides all values via getParameterValues.
- Function – represents a scalar function with parameters that are also Expressions.
- AggregateFunction – represents an aggregate function which can hold a single expression.
- WindowFunction – represents an window function which holds an AggregateFunction (which is also used to represent analytical functions) and a WindowSpecification.
- 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.
- Array – represents an array of expressions, used by the engine in multi-attribute dependent joins.

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 =, >, and so on.
- 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/SIMILAR TO/LIKE_REGEX criteria that compares string values.

The FROM clause contains a list of TableReferences:

- 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 joins with a single root. This latter form is the ANSI preferred style. If you wish all pushdown queries containing joins to be in ANSI style have the capability "useAnsiJoin" return true.

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).

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).

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)

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

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

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.

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.

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

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

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

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.

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.

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 building 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.

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.

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.

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();
```

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.

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.

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 follow these steps:

Procedure 5.3. Write a Visitor

1. 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.

2. 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.

3. 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();
```

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

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 EAP 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.

Dependent joins are a technique used in federation to reduce the cost of cross source joins. Join values from one side of a join are made available to the other side which reduces the number of tuples needed to preform the join. Translators may indicate support for dependent join pushdown via the supportsDependentJoin and supportsFullDependentJoin capabilities. The handling of pushdown dependent join queries can be complicated.

The more simplistic mode of dependent join pushdown is to push only the key (equi-join) values to effectively evaluate a semi-join - the full join will still be processed by the engine after the retrieval. The ordering (if present) and all of the non-dependent criteria constructs on the pushdown command must be honored. The dependent criteria, which will be a Comparison with a Parameter (possibly in Array form), may be ignored in part or in total to retrieve a superset of the tuples requested.

Pushdown key dependent join queries will be instances of Select with the relevant dependent values available via Select.getDependentValues(). A dependent value tuple list is associated to Parameters by id via the Parameter.getDepenentValueId() identifier. The dependent tuple list provide rows that are referenced by the column positions (available via Parameter.getValueIndex()). Care should be taken with the tuple values as they may guaranteed to be ordered, but will be unique with respect to all of the Parameter references against the given dependent value tuple list.

In some scenarios, typically with small independent data sets or extensive processing above the join that can be pushed to the source, it is advantageous for the source to handle the dependent join pushdown. This feature is marked as supported by the supportsFullDependentJoin capability. Here the source is expected to process the command exactly as specified - the dependent join is not optional

Full pushdown dependent join queries will be instances of QueryExpression with the relevant dependent values available via special common table definitions using QueryExpression.getWith(). The independent side of a full pushdown join will appear as a common table WithItem with a dependent value tuple list available via WithItem.getDependentValues(). The dependent value tuples will positionally match the columns defined by WithItem.getColumns(). The dependent value tuple list is not guaranteed to be in any particular order.

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. Translators are allowed to support any subset of the available execution modes.

Table 5.2. Execution Modes

Execution Interface	Command Interface	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.

The org.teiid.translator.ExecutionContext provides a considerable amount of information related to the current execution. An ExecutionContext instance is made available to each Execution. Specific usage is highlighted in this guide where applicable, but you may use any informational getter method as desired. Example usage would include calling ExecutionContext.getRequestId(), ExecutionContext.getSession(), etc. for logging purposes.

An org.teiid.CommandContext is available via the ExecutionContext.getCommandContext() method. The CommandContext contains information about the current user query, including the VDB, the ability to add client warnings - addWarning, or handle generated keys - isReturnAutoGeneratedKeys, returnGeneratedKeys, and getGeneratedKeys.

To see if the user query expects generated keys to be returned, consult the CommandContext.isReturnAutoGeneratedKeys() method. If you wish to return generated keys, you must first create a GeneratedKeys instance to hold the keys with the returnGeneratedKeys method passing the column names and types of the key columns. Only one GeneratedKeys may be associated with the CommandContext at any given time.

The Teiid source meta-hint is used to provide hints directly to source executions via user or transformation queries. See the reference for more on source hints. If specified and applicable, the general and source specific hint will be supplied via the ExecutionContext methods getGeneralHint and getSourceHint. See the source for the OracleExecutionFactory for an example of how this source hint information can be utilized.

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.getBatchSize() method and used as a hint in fetching results from the EIS.

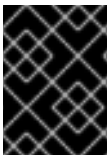
Each execution returns the update count(s) expected by the update command. If possible BatchedUpdates should be executed atomically. The ExecutionContext.isTransactional() method can be used to determine if the execution is already under a transaction.

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.

In some scenarios, a translator needs to execute asynchronously and allow the executing thread to perform other work. To allow asynchronous execution, you should throw a DataNotAvailableExecption during a retrieval method, rather than explicitly waiting or sleeping for the results. The DataNotAvailableException may take a delay parameter or a Date in its constructor to indicate when to poll next for results. Any non-negative delay value indicates the time in milliseconds until the next polling should be performed. The DataNotAvailableException.NO_POLLING exception (or any DataNotAvailableException with a negative delay) can be thrown to indicate that the execution will call ExecutionContext.dataAvailable() to indicate processing should resume.



IMPORTANT

A DataNotAvailableException should not be thrown by the execute method, as that can result in the execute method being called multiple times.

IMPORTANT

Since the execution 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.

A positive retry delay is not a guarantee of when the translator will be polled next. If the DataNotAvailableException is consumed while the engine thinks more work can be performed or there are other shorter delays issued from other translators, then the plan may be re-queued earlier than expected. You should simply rethrow a DataNotAvailableException if your execution is not yet ready. Alternatively the DataNotAvailableException may be marked as strict, which does provide a guarantee that the Execution will not be called until the delay has expired or the given Date has been reached. Using the Date constructor makes the DataNotAvailableException automatically strict. Due to engine thread pool contention, platform time resolution, etc. a strict DataNotAvailableException is not a real-time guarantee of when the next poll for results will occur, only that it will not occur before then.

IMPORTANT

If your ExecutionFactory returns only asynch executions that perform minimal work, then consider having ExecutionFactory.isForkable return false so that the engine knows not to spawn a separate thread for accessing your Execution.

A translator may return instances of ReusableExecutions for the expected Execution objects. There can be one ReusableExecution per query executing node in the processing plan. The lifecycle of a ReusableExecution is different that a normal Execution. After a normal creation/execute/close cycle the ReusableExecution.reset is called for the next execution cycle. This may occur indefinitely depending on how many times a processing node executes its query. The behavior of the close method is no different than a regular Execution, it may not be called until the end of the statement if lobs are detected and any connection associated with the Execution will also be closed. When the user command is finished, the ReusableExecution.dispose() method will be called.

In general ReusableExecutions are most useful for continuous query execution and will also make use

of the ExecutionCotext.dataAvailable() method for Asynchronous Executions. See the Client Developer's Guide for executing continuous statements. In continuous mode the user query will be continuously re-executed. A ReusableExecution allows the same Execution object to be associated with the processing plan for a given processing node for the lifetime of the user query. This can simplify asynch resource management, such as establishing queue listeners. Returning a null result from the next() method ReusableExecution just as with normal Executions indicates that the current pushdown command results have ended. Once the reset() method has been called, the next set of results should be returned again terminated with a null result.

Non batched Insert, Update, Delete commands may have multi-valued Parameter objects if the capabilities shows support for BulkUpdate. Commands with multi-valued Parameters represent multiple executions of the same command with different values. As with BatchedUpdates, bulk operations should be executed atomically if possible.

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.

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.

The main class in the translator implementation is ExecutionFactory. A base class is provided in the Teiid API, so 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 needs to define/override the following elements.

```
package org.teiid.translator.custom;

@Translator(name="custom", description="Connect to My EIS")
public class CustomExecutionFactory extends
ExecutionFactory<MyConnectionFactory, MyConnection> {

    public CustomExecutionFactory() {
    }
}
```

Define the annotation @Translator on extended "ExecutionFactory" class. This annotation defines the name, which is used as the identifier during deployment, and the description of your translator. This name is what you will be using in the VDB and else where in the configuration to refer to this translator.

ConnectionFactory defines the "ConnectionFactory" interface that is defined in resource adapter. This is defined as part of the class definition of the extended "ExecutionFactory" class.

Connection defines the "Connection" interface that is defined in the resource adapter. This is defined as part of class definition of extended "ExecutionFactory" class.

If the translator requires external configuration, that defines ways for the user to alter the behavior of a program, then define an attribute variable in the class and define "get" and "set" methods for that attribute. 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 can automatically interrogate and provide a graphical way to configure your Translator while designing your VDB:

```
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;
}
```

The @TranslatorProperty defines the following metadata that you can set 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 value must 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

Only java primitive (int, boolean), primitive object wrapper (java.lang.Integer), or Enum types are supported as Translator properties. Complex objects are not supported. The default value will be derived from calling the getter method, 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.

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.

Extended Translator Capabiities 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.

Based on types of executions you are supporting, the following methods need to be overridden to provide implementations for their respective return interfaces:

- createResultSetExecution - Override if you are doing read based operation that is returning a rows of results. For ex: select
- createUpdateExecution - Override if you are doing write based operations. For example, insert, update and delete
- createProcedureExecution- Overide if you are doing procedure based operations. For example; stored procedures. This works well for non-relational sources.

You can choose to implement all the execution modes or just what you need.

Override and implement the method getMetadataProcessor(), 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 used by Designer tooling when the Teiid Connection importer is used. A sample MetadataProcessor may look like this:

```
public class MyMetadataProcessor implements MetadataProcessor<Connection>
{

     public void process(MetadataFactory mf, Connection conn) {
            Object somedata = connection.getSomeMetadata();

            Table table = mf.addTable(tableName);
            Column col1 = mf.addColumn("col1",
TypeFacility.RUNTIME_NAMES.STRING, table);
            column col2 = mf.addColumn("col2",
TypeFacility.RUNTIME_NAMES.STRING, table);
     }
}
```

If your MetadataProcessor needs external properties that are needed during the import process, you can define them on MetadataProcessor. For example, to define a import property called "Column Name Pattern", which can be used to filter which columns are defined on the table, can be defined in the code like this:

```
@TranslatorProperty(display="Column Name Pattern",
category=PropertyType.IMPORT, description="Pattern to derive column
names")
public String getColumnNamePattern() {
    return columnNamePattern;
}

public void setColumnNamePattern(String columnNamePattern) {
    this.columnNamePattern = columnNamePattern;
}
```

Note the category type. The configuration property defined in the previous section is different from this one. Configuration properties define the runtime behavior of translator, where as "IMPORT" properties define the metadata import behavior, and aid in controlling what metadata is exposed by your translator.

These properties can be automatically injected through "import" properties set through Designer when using the "Teiid Connection" importer or the properties can be defined under the model construct in the **`vdb.xml`** file, like this:

```
<vdb name="myvdb" version="1">
   <model name="legacydata" type="PHYSICAL">
      <property name="importer.ColumnNamePattern" value="col*"/>
      ....
      <source name = .../>
   </model>
</vdb>
```

There may be times when implementing a custom translator, the built in metadata about your schema is not enough to process the incoming query due to variance of semantics with your source query. To aid this issue, Teiid provides a mechanism called "Extension Metadata", which is a mechanism to define custom properties and then add those properties on metadata object (table, procedure, function, column, index etc.). For example, in a custom translator a table represents a file on disk. This is how you could define such a custom metadata property:

```
public class MyMetadataProcessor implements MetadataProcessor<Connection>
{
     public static final String NAMESPACE = "{http://my.company.corp}";

      @ExtensionMetadataProperty(applicable={Table.class},
datatype=String.class, display="File name", description="File Name",
required=true)
     public static final String FILE_PROP = NAMESAPCE+"FILE";

     public void process(MetadataFactory mf, Connection conn) {
            Object somedata = connection.getSomeMetadata();

            Table table = mf.addTable(tableName);
            table.setProperty(FILE_PROP, somedata.getFileName());

            Column col1 = mf.addColumn("col1",
TypeFacility.RUNTIME_NAMES.STRING, table);
            column col2 = mf.addColumn("col2",
TypeFacility.RUNTIME_NAMES.STRING, table);

     }
}
```

The @ExtensionMetadataProperty defines the following metadata that you can define about your property:

- applicable: Metadata object this is applicable on. This is array of metadata classes like Table.class, Column.class.
- datatype: The java class indicating the data type
- display: Display name of the property
- description: Description about the property
- required: Indicates if the property is a required property

When you define an extension metadata property like above, during the runtime you can obtain the value of that property. If you get the query object which contains 'SELECT * FROM MyTable', MyTable will be represented by an object called "NamedTable":

-

```
for (TableReference tr:query.getFrom()) {
    NamedTable t = (NameTable) tr;
    Table table = t.getMetadataObject();
    String file = table.getProperty(FILE_PROP);
    ..
}
```

Now you have accessed the file name you set during the construction of the Table schema object, and you can use this value however you seem feasible to execute your query. With the combination of built in metadata properties and extension metadata properties you can design and execute queries for a variety of sources.

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 EAP's profile to add the custom context. Teiid uses Log4J as its underlying logging system.

If you need to trace any exception use org.teiid.translator.TranslatorException class.

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.

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:

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

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.

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.

An 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 via the ExecutionContext.keepAlive method.

When the "keepAlive" flag is set, then the execution object is only closed when user's statement is closed:

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

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.

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 necessary methods to specify which capabilities your translator supports. Consult the debug log of query planning (set showplan debug) to see if the pushdown you desire requires additional capabilities.

Note capabilities are determined and cached for the lifetime of the translator. Capabilities based on connection/user are not supported.

These capabilities can be specified in the ExecutionFactory class.

CHAPTER 6. EXTENDING THE EXECUTION FACTORY CLASS

6.1. EXTENDING THE EXECUTIONFACTORY CLASS

A custom translator must extend the **`org.teiid.translator.ExecutionFactory`** class to connect and query a data source. This extended class must provide a constructor with no arguments that can be constructed using Java reflection libraries.

The following is an example constructor:

```
package org.teiid.translator.custom;

@Translator(name="custom", description="Connect to My EIS")
public class CustomExecutionFactory extends
ExecutionFactory<MyConnectionFactory, MyConnection> {

    public CustomExecutionFactory() {
    }
}
```

Specify the annotation **`@Translator`** on the extended "ExecutionFactory" class. This annotation defines the name and description of your translator, and is also used as an identifier during deployment. This is the name you would be using in the VDB and elsewhere in the configuration to refer to this translator.

`MyConnectionFactory` specifies the type of ConnectionFactory interface that is expected from the associated resource adapter. This is required as part of the class definition when extending the **`ExecutionFactory`** class.

`MyConnection` specifies the type of Connection interface that is expected from the associated resource adapter. This is required as part of class definition when extending the **`ExecutionFactory`** class.

6.2. CONFIGURATION PROPERTIES

If the translator requires configurable properties then:

1. define a variable for every property as an attribute in the extended **`ExecutionFactory`** class,
2. define "get" and "set" methods for each attribute,
3. and 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, JBoss Data Virtualization will automatically interrogate and provide a graphical way to configure your Translator while designing your VDB.

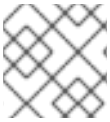
```
private String foo = "blah";
@TranslatorProperty(display="Foo property", description="description about
Foo")
public String getFoo()
{
   return foo;
```

```
}

public void setFoo(String value)
{
   return this.foo = value;
}
```

Only java primitive (**int**, **boolean**), 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 - the display name of the property.
- description - a description about the property.
- required - specifies that the property is required.
- advanced - an advanced property (a default value must be provided).
- masked - tools need to mask the property, that is, do not show it in plain text. Used for passwords.



NOTE

A property can not be "advanced" and "required" at the same time.

6.3. INITIALIZING THE TRANSLATOR

Override and implement the **start()** method if your translator needs to do any initialization before it is used by the JBoss Data Virtualization engine. This method must also call **super.start()** to perform any initialization required by the superclass. This method is called by JBoss Data Virtualization once all the configuration properties are injected into the class.

6.4. EXTENDED TRANSLATOR CAPABILITIES

There are various methods, typically beginning with the method signature **supports**, that specify translator capabilities. These methods need to be overridden to describe the execution capabilities of the Translator. See Section 10.1, “Translator Capabilities” for more information about these methods.

6.5. EXECUTION (AND SUB-INTERFACES)

Based on types of executions you are supporting, the following methods need to be overridden to provide implementations for their respective return interfaces.

- **createResultSetExecution** - Override if you are doing read based operation that is returning rows of results. For example, select.
- **createUpdateExecution** - Override if you are doing write based operations. For example, insert, update, delete.

- **`createProcedureExecution`** - Override if you are doing procedure based operations. For example, stored procedures. This works well for non-relational sources.

You can choose to implement all the execution modes or only what you need. Refer to Section A.1, “Execution Modes” for more information.

6.6. METADATA

You can override and implement the method getMetadataProcessor(), in order 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 used by Designer tooling when the Teiid Connection importer is used. A sample MetadataProcessor may look like this:

```
  public class MyMetadataProcessor implements
MetadataProcessor<Connection> {

     public void process(MetadataFactory mf, Connection conn) {
            Object somedata = connection.getSomeMetadata();

            Table table = mf.addTable(tableName);
            Column col1 = mf.addColumn("col1",
TypeFacility.RUNTIME_NAMES.STRING, table);
            column col2 = mf.addColumn("col2",
TypeFacility.RUNTIME_NAMES.STRING, table);

            //add a pushdown function that can also be evaluated in the
engine
            Method method = ...
            Function f = mf.addFunction("func", method);

            //add a pushdown aggregate function that can also be evaluated
in the engine
            Method aggMethod = ...
            Function af = mf.addFunction("agg", aggMethod);
            af.setAggregateAttributes(new AggregateAttributes());
            ...
     }
}
```

If your MetadataProcessor needs external properties that are needed during the import process, you can define them on MetadataProcessor. For example, to define a import property called "Column Name Pattern", which can be used to filter which columns are defined on the table, you can add it like this:

```
@TranslatorProperty(display="Column Name Pattern",
category=PropertyType.IMPORT, description="Pattern to derive column
names")
public String getColumnNamePattern() {
    return columnNamePattern;
}

public void setColumnNamePattern(String columnNamePattern) {
    this.columnNamePattern = columnNamePattern;
}
```

Note the category type. The configuration property defined in the previous section is different from this one. Configuration properties define the runtime behavior of translator, where as "IMPORT" properties define the metadata import behavior, and aid in controlling what metadata is exposed by your translator.

These properties can be automatically injected through "import" properties set through Designer when using the "Teiid Connection" importer or the properties can be defined under the model construct in the vdb.xml file, like

```
<vdb name="myvdb" version="1">
   <model name="legacydata" type="PHYSICAL">
      <property name="importer.ColumnNamePattern" value="col*"/>
      ....
      <source name = .../>
   </model>
</vdb>
```

There may be times when implementing a custom translator, the built in metadata about your schema is not enough to process the incoming query due to variance of semantics with your source query. To aid this issue, Teiid provides a mechanism called "Extension Metadata", which is a mechanism to define custom properties and then add those properties on metadata object (table, procedure, function, column, index etc.). For example, in my custom translator a table represents a file on disk. I could define a extension metadata property like this:

```
public class MyMetadataProcessor implements MetadataProcessor<Connection>
{
     public static final String NAMESPACE = "{http://my.company.corp}";

      @ExtensionMetadataProperty(applicable={Table.class},
datatype=String.class, display="File name", description="File Name",
required=true)
     public static final String FILE_PROP = NAMESAPCE+"FILE";

     public void process(MetadataFactory mf, Connection conn) {
            Object somedata = connection.getSomeMetadata();

            Table table = mf.addTable(tableName);
            table.setProperty(FILE_PROP, somedata.getFileName());

            Column col1 = mf.addColumn("col1",
TypeFacility.RUNTIME_NAMES.STRING, table);
            column col2 = mf.addColumn("col2",
TypeFacility.RUNTIME_NAMES.STRING, table);

     }
}
```

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

- applicable: Metadata object this is applicable on. This is array of metadata classes like Table.class, Column.class.
- datatype: The java class indicating the data type
- display: Display name of the property

- description: Description about the property
- required: Indicates if the property is a required property

When you define an extension metadata property like above, during the runtime you can obtain the value of that property. If you get the query object which contains 'SELECT * FROM MyTable', MyTable will be represented by an object called "NamedTable".

```
for (TableReference tr:query.getFrom()) {
    NamedTable t = (NameTable) tr;
    Table table = t.getMetadataObject();
    String file = table.getProperty(FILE_PROP);
    ..
}
```

Now you have accessed the file name you set during the construction of the Table schema object, and you can use this value however you seem feasible to execute your query. With the combination of built in metadata properties and extension metadata properties you can design and execute queries for a variety of sources.

6.7. LOGGING

JBoss Data Virtualization provides the **`org.teiid.logging.LogManager`** class for logging purposes, based on the Apache Log4j logging services.

Logging messages will be sent automatically to the main JBoss Data Virtualization logs. You can customize logging by editing the corresponding subsystem in the server configuration file or via the Management Console.

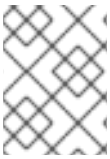
6.8. EXCEPTIONS

When throwing exceptions in translator code, use the **`org.teiid.translator.TranslatorException`** class.

6.9. DEFAULT NAME

You can define a default instance of your Translator by defining the annotation **`@Translator`** on the **`ExecutionFactory`**. After deployment, a default instance of this Translator can be used by any VDB by referencing it by this name in its **`vdb.xml`** configuration file.

A 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 everywhere else in the JBoss Data Virtualization configuration to refer to this translator.



NOTE

The translator created here is only available in the scope of the VDB - it is not available to the whole JBoss Data Virtualization instance.

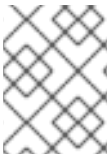
6.10. OBTAINING CONNECTIONS

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

6.11. RELEASING CONNECTIONS

Connections are only used for the lifetime of the request. When the request completes, the `closeConnection()` method is called on the `ExecutionFactory`. You must override this method to close the connection properly.

If the resource adapter is JEE JCA Connector based, connection pooling is automatically provided.



NOTE

Red Hat recommends the use of connection pooling when a connection is stateful or when connections are expensive to create.

CHAPTER 7. EXTENDING THE JDBC TRANSLATOR

7.1. EXTENSIONS

New custom Translators can be created by extending the JDBC Translator. This is one of the most common use-cases for custom Translator development and is often done to add support for JDBC drivers and database versions.

To design a JDBC Translator for any relational database management system (RDBMS) that is not already supported by JBoss Data Virtualization, 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 7.1. 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.

7.2. CAPABILITIES EXTENSION

This extension must override the methods that begin with "supports" that describe translator capabilities. Refer to Section 10.3, “Available 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.

7.3. 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 is to 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 will 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.

7.4. 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.

7.5. ADDING FUNCTION SUPPORT

Refer to the section on user defined functions for adding new functions to JBoss Data Virtualization. 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 JBoss Data Virtualization function if the JDBC data source supports the same syntax as JBoss Data Virtualization. The built-in SQL translation will translate most functions as: "function(arg1, arg2, ...)".

7.6. USING FUNCTION MODIFIERS

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 7.2. Common Modifiers

Modifier	Description
AliasModifier	Handles 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.

7.7. INSTALLING EXTENSIONS

Once you have developed an extension to the JDBC translator, you must install it into the JBoss Data Virtualization server. The process of packaging or deploying the extended JDBC translators is exactly as any other translator. Since the RDBMS is accessible already through its JDBC driver, there is no need to develop a resource adapter for this source as JBoss EAP provides a wrapper JCA connector (DataSource) for any JDBC driver.

CHAPTER 8. TRANSLATOR DEVELOPMENT AND LARGE OBJECTS

8.1. DATA TYPES

JBoss Data Virtualization 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 an "xml document". Columns modeled as a BLOB, CLOB, or XML are treated similarly by the translator framework to support memory-safe streaming.

8.2. WHY USE LARGE OBJECT SUPPORT?

JBoss Data Virtualization allows a Translator to return a large object through the translator API by 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.

8.3. 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 can then, for example, appear in client results, in user defined functions, or be sent to other translators.

An 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 using the **`ExecutionContext.keepAlive()`** method.

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

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

8.4. 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.

CHAPTER 9. OTHER CONSIDERATIONS FOR TRANSLATOR DEVELOPMENT

9.1. CACHING API

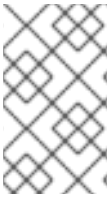
Translators may contribute cache entries to the result set cache by the use of the **CacheDirective** object. Translators wishing to participate in caching should return a **CacheDirective** from the **ExecutionFactory.getCacheDirective** method, which is called prior to execution. The commands passed to **getCacheDirective** will have already been vetted to ensure that the results are eligible for caching. For example update commands or commands with pushed dependent sets will not be eligible for caching.

If the translator returns null for the **CacheDirective** , which is the default implementation, the engine will not cache the translator results beyond the current command. It is up to your custom translator or custom delegating translator to implement your desired caching policy.

NOTE

In special circumstances where the translator has performed its own caching, it can indicate to the engine that the results should not be cached or reused by setting the **Scope** to **Scope.NONE** .

The returned **CacheDirective** will be set on the **ExecutionContext** and is available via the **ExecutionContext.getCacheDirective()** method. Having **ExeuctionFactory.getCacheDirective** called prior to execution allows the translator to potentially be selective about which results to even attempt to cache. Since there is a resource overhead with creating and storing the cached results it may not be desirable to attempt to cache all results if it is possible to return large results that have a low usage factor. If you are unsure about whether to cache a particular command result you may return an initial **CacheDirective** then change the **Scope** to **Scope.NONE** at any time prior to the final cache entry being created and the engine will give up creating the entry and release its resources.



NOTE

If you plan on modifying the **CacheDirective** during execution, return a new instance from the **ExecutionFactory.getCacheDirective** call, rather than returning a shared instance.

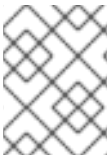
The **CacheDirective** readAll Boolean field is used to control whether the entire result should be read if not all of the results were consumed by the engine. If readAll is false then any partial usage of the result will not result in it being added as a cache entry. Partial use is determined after any implicit or explicit limit has been applied. The other fields on the **CacheDirective** object map to the cache hint options . See the table below for the default values for all options.

option	default
scope	Session
ttl	rs cache ttl

option	default
readAll	true
updatable	true
prefersMemory	false

9.2. DEPENDENT JOIN PUSHDOWN

Dependent joins are a technique used in federation to reduce the cost of cross source joins. Join values from one side of a join are made available to the other side which reduces the number of tuples needed to preform the join. Translators may indicate support for dependent join pushdown via the supportsDependentJoin capability. The handling of pushdown dependent join queries can be quite complicated. The ordering (if present) and all of the non-dependent criteria constructs on the pushdown command must be honored, but if needed the dependent criteria, which will be a **`Comparison`** with a **`Parameter`**, may be ignored in part or in total. Pushdown dependent join queries will be instances of **`Select`** with the relevant dependent sets available via **`Select.getDependentSets()`**. The dependent set is associated to Parameters by id via the **`Parameter.getDepenentValueId()`** identifier. The dependent set tuple iterators provide rows that are referenced by the column positions (available via **`Parameter.getValueIndex()`**) on the dependent join Comparison criteria right expression. Care should be taken with the tuple values as they may guaranteed to be unique or ordered.



NOTE

There is no reference implementation of this functionality as all built-in translators rely on the engine to handle breaking up dependent joins into simpler queries.

9.3. DELEGATING TRANSLATOR

In some instances, you may wish to extend multiple translators with the same functionality. Rather than create separate subclasses for each extension, functionality that is common to multiple extensions can be added to a subclass of **`BaseDelegatingExecutionFactory`**. Within this subclass, delegation methods can be overridden to perform the common functionality.

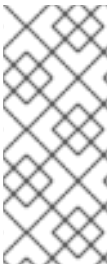
```
@Translator(name="custom-delegator")
public class MyTranslator extends BaseDelegatingExecutionFactory<Object,
Object> {

        @Override
        public Execution createExecution(Command command,
                        ExecutionContext executionContext,
RuntimeMetadata metadata,
                        Object connection) throws TranslatorException {
                if (command instanceof Select) {
                        //modify the command or return a different
execution
                        ...

                }
```

```
                //the super call will be to the delegate instance
                return super.createExecution(command, executionContext,
metadata, connection);
        }
        ...
}
```

You will bundle and deploy your custom delegating translator like any other custom translator development. To use your delegating translator in a VDB, you define a translator override that wires in the delegate.

```
<translator type="custom-delegator" name="my-translator">

     <property value="delegateName" name="name of the delegate instance"/>

     <!-- any custom properties you may have on your custom translator -->

</translator>
```

From the previous example the translator type is custom-delegator. Now my-translator can be used as a translator-name on a source and will proxy all calls to whatever delegate instance you assign.



NOTE

The delegate instance can be any translator instance whether configured by its own translator entry or only the name of a standard translator type. Using a **`BaseDelegatingExecutionFactory`** by default means that standard override translator property settings on your instance will have no effect, since the underlying delegate is called instead.

You may also wish to use a different class hierarchy and instead make your custom translator only implement **`DelegatingExecutionFactory`** instead.

9.4. ADDING DEPENDENT MODULES

Add a MANIFEST.MF file in the META-INF directory, and the core API dependencies for resource adapter with the following line.

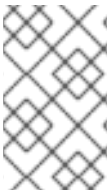
```
Dependencies: org.jboss.teiid.common-core,org.jboss.teiid.api,javax.api
```

If your translator depends upon any other third party jar files, ensure a module exists and add the module name to the above MANIFEST.MF file.

CHAPTER 10. TRANSLATOR CAPABILITIES

10.1. 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 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.



NOTE

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.

10.2. TRANSLATOR CAPABILITIES

During translator development, you can define three different types of property sets that can help customize the behavior of the translator.

On the "ExecutionFactory" class a translator developer can define any number of "getter/setter" methods with the @TranslatorProperty annotation. These properties (also referred to a execution properties) can be used for extending the capabilities of the translator. It is important to define default values for all these properties, as these properties are being defined to change the default behavior of the translator. If needed, the values for these properties are supplied in "vdb.xml" file during the deploy time when the translator is used to represent vdb's model. Here is an example:

```
@TranslatorProperty(display="Copy LOBs",description="If true, returned
LOBs will be copied, rather than streamed from the source",advanced=true)
public boolean isCopyLobs() {
    return copyLobs;
}

public void setCopyLobs(boolean copyLobs) {
    this.copyLobs = copyLobs;
}
```

At runtime, you can define these properties in the vdb.xml file like this:

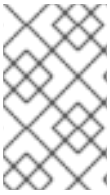
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="vdb" version="1">
    <model name="PM1">
        <source name="connector" translator-name="my-translator-override"
/>
    </model>
    <translator name="my-translator-override" type="my-translator">
        <property name="CopyLobs" value="true" />
    </translator>
</vdb>
```

If a translator is defining schema information based on the physical source (i.e. implementing getMetadata method on ExecutionFactory) it is connected to, then import properties provide a way to customize the behavior of the import process. For example, in the JDBC translator users can exclude certain tables that match a regular expression etc. To define a import property, the @TranslatorPropery annotation is used on any getter/setter method on the "ExecutionFactory" class or any class that implements the "MetadataProcessor" interface, with category property defined as "PropertyType.IMPORT":

```
@Translator(name = "my-translator", description = "My Translator")
public class MyExecutionFactory extends
ExecutionFactory<ConnectionFactory, MyConnection> {
...
    public MetadataProcessor<C> getMetadataProcessor() {
        return MyMetadataProcessor();
    }
}

public MyMetadataProcessor implements MetadataProcessor<MyConnection> {

    public void process(MetadataFactory metadataFactory, MyConnection
connection) throws TranslatorException{
        // schema generation code here
    }

    @TranslatorProperty(display="Header Row Number",
category=PropertyType.IMPORT, description="Row number that contains the
header information")
    public int getHeaderRowNumber() {
        return headerRowNumber;
    }

    public void setHeaderRowNumber(int headerRowNumber) {
        this.headerRowNumber = headerRowNumber;
    }
}
```

This is how you use import properties with a vdb.xml file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="vdb" version="1">
    <model name="PM1">
        <property name="importer.HeaderRowNumber" value="12"/>
        <source name="connector" translator-name="my-translator" />
    </model>
</vdb>
```



NOTE

When properties are defined using the annotation mechanism and also when you use the "Teiid Connection" importer in the Designer, these properties will automatically show up in the wizard's relevant input field.

During the execution of the command in translator, a translator is responsible to convert Teiid supplied SQL command into data source specific query. Most of times this conversion is not a trivial task can be

converted from one form to another. There are many cases built-in metadata is not sufficient and additional metadata about source is useful to form a request to the underlying physical source system. Extension Metadata Properties one such mechanism to fill the gap in the metadata. These can be defined specific for a given translator.

A translator is a plugin, that is communicating with Teiid engine about it's source with it's metadata. Metadata in this context is definitions of Tables, Columns, Procedures, Keys etc. This metadata can be decorated with additional custom metadata and fed to Teiid query engine. Teiid query engine keeps this extended metadata intact along with its schema objects, and when a user query is submitted to the the translator for execution, this extended metadata can be retrieved for making decisions in the translator code.

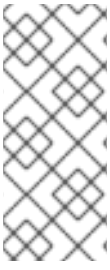
Extended properties are defined using annotation class called @ExtensionMetadataProperty on the fields in your "MetadataProcessor" or "ExcutionFactory" classes.

For example, say translator requires a "encoding" property on Table, to do the correct un-marshaling of data, this property can be defined like this:

```
public class MyMetadataProcessor implements
MetadataProcessor<MyConnection> {
    public static final String URI = "
{http://www.teiid.org/translator/mytranslator/2014}";

    @ExtensionMetadataProperty(applicable=Table.class,
datatype=String.class, display="Encoding", description="Encoding",
required=true)
    public static final String ENCODING = URI+"encode";

    public void process(MetadataFactory mf, FileConnection conn) throws
TranslatorException {
        ..
        Table t = mf.addTable(tableName);
        t.setProperty(ENCODING, "UTF-16");

        // add columns etc.
        ..
    }
}
```

Now during the execution, on the COMMAND object supplied to the "Execution" class, user can do this:

```
Select select = (Select)command;
NamedTable tableReferece = select.getFrom().get(0);
Table t = tableReference.getMetadataObject();
String encoding = t.getProperty(MyMetadataProcessor.ENCODING, false);

// use the encoding value as needed to marshal or unmarshal data
```



NOTE

When extended properties are defined using the annotation mechanism, when using "Teiid Connection" importer in the Designer, you do not need to define the "Metadata Extension Defn" in designer and register to use with your model, the required definitions are automatically downloaded and configured to use. (This feature is not available in current Designer version).

10.3. AVAILABLE CAPABILITIES

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

Table 10.1. Available Capabilities

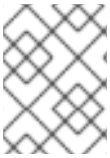
Capability	Requires	Description
SelectDistinct		Translator can support SELECT DISTINCT in queries.
SelectExpression		Translator can support SELECT of more than column references.
AliasedTable		Translator can support Tables in the FROM clause that have an alias.
InnerJoins		Translator can support inner 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.
DependentJoins	Base join and criteria support	Translator supports key set dependent join pushdown (see Section 9.2, "Dependent Join Pushdown"). When set, the MaxDependentInPredicates and MaxInCriteriaSize values are not used by the engine, rather all independent values are made available to the pushdown command.
SubqueryInOn	Join and base subquery support, such as ExistsCriteria	Translator can support subqueries in the ON clause. Defaults to true.
InlineViews	AliasedTable	Translator can support a named subquery in the FROM clause.
BetweenCriteria		Not currently used - between criteria is rewritten as compound comparisons.
CompareCriteriaEquals		Translator can support comparison criteria with the operator "=".
CompareCriteriaOrdered		Translator can support comparison criteria with the operator ">" or "<".

Capability	Requires	Description
LikeCriteria		Translator can support LIKE criteria.
LikeCriteriaEscapeCharacter	LikeCriteria	Translator can support LIKE criteria with an ESCAPE character clause.
SimilarTo		Translator can support SIMILAR TO criteria.
LikeRegexCriteria		Translator can support LIKE_REGEX criteria.
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		Translator can support a quantified comparison criteria using the SOME or ANY quantifier.
OnlyLiteralComparison		Translator if only Literal comparisons (equality, ordered, like, etc.) are supported for non-join conditions.
Convert(int fromType, int toType)		Used for fine grained control of convert/cast pushdown. The **`ExecutionFactory.getSupportedFunctions()`** should contain **`SourceSystemFunctions.CONVERT`**. This method can then return false to indicate a lack of specific support. See **`TypeFacility.RUNTIME_CODES`** for the possible type codes. The engine will does not care about an unnecessary conversion where fromType == toType. By default lob conversion is disabled.
OrderBy		Translator can support the ORDER BY clause in queries.

Capability	Requires	Description
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 explicit 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 rewritten 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

Capability	Requires	Description
Except		Translator supports Except
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	Translator can support non-column reference grouping expressions.
InsertWithQueryExpression		Translator supports INSERT statements with values specified by a QueryExpression.
BatchedUpdates		Translator supports a batch of INSERT, UPDATE and DELETE commands to be executed together.
BulkUpdate		Translator supports updates with multiple value sets
CommonTableExpressions		Translator supports the WITH clause.
ElementaryOlapOperations		Translator supports window functions and analytic functions RANK, DENSE_RANK, and ROW_NUMBER.
WindowOrderByWithAggregates	ElementaryOlapOperations	Translator supports windowed aggregates with a window order by clause.
WindowDistinctAggregates	ElementaryOlapOperations, AggregatesDistinct	Translator supports windowed distinct aggregates.
AdvancedOlapOperations	ElementaryOlapOperations	Translator supports aggregate conditions.
OnlyFormatLiterals	function support for a parse/format function and an implementation of the supportsFormatLiteral method.	Translator supports only literal format patterns that must be validated by the supportsFormatLiteral method
FormatLiteral(String literal, Format type)	OnlyFormatLiterals	Translator supports the given literal format string.

Capability	Requires	Description
ArrayType		Translator supports the push down of array values.
OnlyCorrelatedSubqueries	CorrelatedSubqueries	Translator ONLY supports correlated subqueries. Uncorrelated scalar and exists subqueries will be pre-evaluated prior to push-down.
SelectWithoutFrom	SelectExpressions	Translator supports selecting values without a FROM clause, such as SELECT 1.



NOTE

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

10.4. 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".

10.5. SCALAR FUNCTIONS

The method **`ExecutionFactory.getSupportedFunctions()`** can be used to specify which scalar and aggregate functions the Translator supports. The set of possible functions is based on the set of functions supported by JBoss Data Virtualization. This set can be found in the JBoss Data Virtualization Reference Guide. If the Translator states that it supports a function, it must support all type combinations and overloaded forms of that function.

There are also some standard operators that can 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 evaluated in JBoss Data Virtualization, are simple aliases to other functions, or are rewritten to a more standard expression.

A translator may also indicate support for scalar functions that are intended for pushdown evaluation by that translator, but are not registered as user defined functions via a model/schema. These pushdown functions are reported to the engine via the **`ExecutionFactory.getPushDownFunctions()`** list as **`FunctionMethod`** metadata objects. The **`FuncitonMethod`** representation allow the translator to control all of the metadata related to the function, including type signature, determinism, varargs, etc. The simplest way to add a pushdown function is with a call to **`ExecutionFactory.addPushDownFunction`**:

```
FunctionMethod addPushDownFunction(String qualifier, String name, String
returnType, String...paramTypes)
```

This resulting function will be known as sys.qualifier.name, but can be called with name only as long as the function name is unique. The returned **`FunctionMethod`** object may be further manipulated depending upon the needs of the source. An example of adding a custom concat vararg function in an **`ExecutionFactory`** subclass:

```
public void start() throws TranslatorException {
  super.start();
  FunctionMethod func = addPushDownFunciton("oracle", "concat", "string",
"string", "string");
  func.setVarArgs(true);
  ...
}
```

10.6. 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.

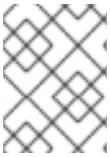
The method **`ExecutionFactory.getMaxDependentInPredicates()`** is used to specify the maximum number of IN predicates (of at most MaxInCriteriaSize) that can be passed as part of a dependent join. For example if there are 10000 values to pass as part of the dependent join and a MaxInCriteriaSize of 1000 and a MaxDependentInPredicates setting of 5, then the dependent join logic will form two source queries each with 5 IN predicates of 1000 values each combined by OR.

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.

10.7. 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

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

10.8. NULL ORDERING

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

The method `ExecutionFactory.getCollation()` specifies the default collation. If you set it to a value that does not match the collation locale defined by org.teiid.collationLocale, then some ordering may not be pushed down.

CHAPTER 11. PACKAGING AND DEPLOYING THE TRANSLATOR

11.1. PACKAGING

Once the "ExecutionFactory" class is implemented, package it in a JAR file. Then add the following named file in "META-INF/services/org.teiid.translator.ExecutionFactory" with contents specifying the name of your main Translator file. Note that, the name must exactly match to above. This is Java's standard service loader pattern. This will register the Translator for deployment when the JAR is deployed.

```
org.teiid.translator.custom.CustomExecutionFactory
```

11.2. TRANSLATOR DEPLOYMENT OVERVIEW

A translator JAR file can be deployed either as a JBoss module or by direct JAR deployment.

11.3. MODULE DEPLOYMENT

Create a module under the "modules" directory and define the translator name and module name in the teiid subsystem in **`standalone.xml`** file or **`domain.xml`** file and restart the server. The dependent JBoss Data Virtualization or any other Java class libraries must be defined in module.xml file of the module. For production profiles this is recommended.

Example 11.1. Example `module.xml` file

The following example is the **`module.xml`** file provided for the Salesforce translator. This file is located in the ***`EAP_HOME`*`/docs/teiid/datasources/salesforce/modules/org/springframework/spring/main`** directory.

```
<module xmlns="urn:jboss:module:1.0" name="org.springframework.spring">
  <resources>
    <resource-root path="spring-beans.jar"/>
    <resource-root path="spring-context.jar"/>
    <resource-root path="spring-core.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
  </dependencies>
</module>
```

11.4. JAR DEPLOYMENT

For development time or quick deployment you can deploy the translator JAR using the Management CLI or AdminShell or Management Console. When you deploy dependencies in JAR form to JBoss Data Virtualization, Java libraries and any other third-party libraries must be defined under META-INF/MANIFEST.MF file.

Example 11.2. Example `MANIFEST.mf` file

The following example is the **/META-INF/MANIFEST.mf** file provided in the Salesforce translator JAR file, ***EAP_HOME*/modules/system/layers/dv/org/jboss/teiid/translator/salesforce/main/translator-salesforce[VERSION].jar**.

```
Manifest-Version: 1.0
Build-Timestamp: Wed, 30 Oct 2013 07:24:14 -0400
Implementation-Title: Salesforce Translator
Implementation-Version: 8.4.1-redhat-4
Os-Version: 2.6.32-358.18.1.el6.x86_64
Built-By: mockbuild
Specification-Vendor: JBoss by Red Hat
Created-By: Apache Maven
Os-Name: Linux
Implementation-URL: http://www.jboss.org/ip-parent/teiid-parent/connec
 tors/translator-salesforce
Java-Vendor: Sun Microsystems Inc.
Implementation-Vendor: JBoss by Red Hat
Scm-Revision: d19b010480ee51efe9f82956d8775e1a604657fd
Implementation-Vendor-Id: org.jboss.teiid.connectors
Scm-Url: http://github.com/jboss/jboss-parent-pom/ip-parent/teiid-pare
 nt/connectors/translator-salesforce
Build-Jdk: 1.6.0_24
Java-Version: 1.6.0_24
Scm-Connection: scm:git:git@github.com:jboss/jboss-parent-pom.git/ip-p
 arent/teiid-parent/connectors/translator-salesforce
Os-Arch: i386
Specification-Title: Salesforce Translator
Specification-Version: 8.4.1-redhat-4
Archiver-Version: Plexus Archiver
```

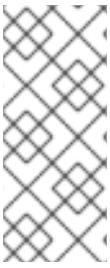
CHAPTER 12. USER DEFINED FUNCTIONS

12.1. USER DEFINED FUNCTIONS

You can extend the JBoss Data Virtualization scalar function library by creating User Defined Functions (UDFs), as well as User Defined Aggregate Functions (UDAFs).

The following are used to define a UDF:

- *Function Name* - When you create the function name, keep these requirements in mind:
  - You cannot overload existing JBoss Data Virtualization functions.
  - The function name must be unique among user-defined functions in its model 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 cannot contain the '.' character.
  - The function name cannot exceed 255 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. If REQUIRED is used, then user must extend the Translator for the source and add this function to its pushdown function library.
- *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. Defaults to false. It is important to mark the function as deterministic if it returns the same value for the same inputs as this will lead to better performance. See also the Relational extension boolean metadata property "deterministic" and the DDL OPTION property "determinism".



NOTE

If using the pushdown UDF in Teiid Designer, the user must create a source function on the source model, so that the parsing will work correctly. 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.

12.2. SUPPORT FOR NON-PUSHDOWN USER DEFINED FUNCTIONS

To define a non-pushdown function, a Java function must be provided that matches the metadata supplied either in the Teiid Designer or Dynamic VDB defined metadata. User Defined Function (or UDF) and User Defined Aggregate Function (or UDAF) may be called at runtime like any other function or aggregate function respectively.

12.2.1. Non-Pushdown UDF Metadata in Teiid Designer

You can create a user-defined function on any VDB in a view model. To do so, create a function as a base table. Make sure you provide the JAVA code implementation details in the properties dialog for the UDF.

12.2.2. Non-Pushdown UDF Metadata for Dynamic VDBs

When defining the metadata using DDL in the Dynamic VDBs, user can define a UDF or UDAF (User Defined Aggregate Function) as shown below.

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="VIRTUAL">
         <metadata type="DDL"><![CDATA[
             CREATE VIRTUAL FUNCTION celsiusToFahrenheit(celsius decimal)
RETURNS decimal OPTIONS (JAVA_CLASS 'org.something.TempConv',  JAVA_METHOD
'celsiusToFahrenheit');
             CREATE VIRTUAL FUNCTION sumAll(arg integer) RETURNS integer
OPTIONS (JAVA_CLASS 'org.something.SumAll',  JAVA_METHOD 'addInput',
AGGREGATE 'true', VARARGS 'true', "NULL-ON-NULL" 'true');]]> </metadata>
    </model>
</vdb>
```

You must create a Java method that contains the function's logic. This Java method should accept the necessary arguments, which JBoss Data Virtualization will pass to it at runtime, and function should return the calculated or altered value.

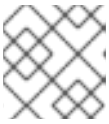
Refer to *Red Hat JBoss Data Virtualization Development Guide: Reference Material* for more information about DDL Metadata and options related to functions defined via DDL.

12.2.3. Coding Non-Pushdown Functions

12.2.3.1. UDF Coding

The following are requirements for coding User Defined Functions (UDFs):

- The Java class containing the function method must be defined public.



NOTE

You can declare multiple user defined functions for a given class.

- The function method must be public and static.

Example 12.1. Sample UDF 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;
    }
}
```

12.2.3.2. UDAF Coding

The following are requirements for coding User Defined Aggregate Functions (UDAFs):

- The Java class containing the function method must be defined public and extend **`org.teiid.UserDefinedAggregate`**.
- The function method must be public.

Example 12.2. Sample UDAF Code

```
package org.something;

public static class SumAll implements UserDefinedAggregate<Integer> {

        private boolean isNull = true;
        private int result;

        public void addInput(Integer... vals) {
                isNull = false;
                for (int i : vals) {
                        result += i;
                }
        }

        @Override
        public Integer getResult(org.teiid.CommandContext
commandContext) {
                if (isNull) {
                        return null;
                }
                return result;
        }

        @Override
        public void reset() {
                isNull = true;
                result = 0;
```

```
        }

}
```

12.2.3.3. Coding: Other Considerations

The following are additional considerations when coding UDFs or UDAFs:

- Number of input arguments and types must match the function metadata defined in Section 12.1, "User Defined Functions".
- Any exception can be thrown, but JBoss Data Virtualization will throw 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, subject, the VDB, the session id, etc. This **`CommandContext`** parameter should not be declared in the function metadata.

Example 12.3. Sample `CommandContext` Usage

```
package org.something;

public class SessionInfo
{
   /**
   * @param context
   * @return the created Timestamp
   */
   public static Timestamp sessionCreated(CommandContext context)
   {
      return new Timestamp(context.getSession().getCreatedTime());
   }
}
```

The corresponding user-defined function would be declared as **`Timestamp sessionCreated()`**.

12.2.3.4. Post Coding Activities

1. After coding the functions, compile the Java code into a Java Archive (JAR) file.
2. Create a JBoss EAP module (**`module.xml`**) accompanying the JAR file in the ***`EAP_HOME`*`/modules/`** directory.
3. Add the module dependency to the ***`DATABASE`*`-vdb.xml`** file as shown in the example below.

```
<vdb name="{vdb-name}" version="1">
    <property name ="lib" value ="{module-name}"></property>
     ...
</vdb>
```

The **`lib`** property value may contain a space delimited list of module names if more than one

dependency is needed.

NOTE

Alternatively, when using a VDB created with Teiid Designer (***DATABASE*.vdb**), the JAR file may be placed in your VDB under the **`/lib`** directory. It will be added automatically to the VDB classloader.

12.3. SOURCE SUPPORTED FUNCTIONS

While JBoss Data Virtualization 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 is 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 order to configure JBoss Data Virtualization to push the above function evaluation to Oracle, you can either: extend the translator in Java, define the function as a pushdown function via Teiid Designer, or, for dynamic VDBs, define it in the VDB.

12.3.1. Defining a Source Supported Function by Extending the Translator

The ExecutionFactory.getPushdownFunctions method can be used to describe functions that are valid against all instances of a given translator type. The function names are expected to be prefixed by the translator type, or some other logical grouping, e.g. salesforce.includes. The full name of the function once imported into the system will qualified by the SYS schema, e.g. SYS.salesforce.includes.

Any functions added via these mechanisms do not need to be declared in ExecutionFactory.getSupportedFunctions. Any of the additional handling, such as adding a FunctionModifier, covered above is also applicable here. All pushdown functions will have function name set to only the simple name. Schema or other qualification will be removed. Handling, such as function modifiers, can check the function metadata if there is the potential for an ambiguity.

To extend the Oracle Connector:

- *Required* - extend the OracleExecutionFactory and add SCORE and CONTAINS as supported pushdown functions by either overriding or adding additional functions in "getPushDownFunctions" method. For this example, we'll call the class MyOracleExecutionFactory. Add the **`org.teiid.translator.Translator`** annotation to the class, e.g. **`@Translator(name="myoracle")`**

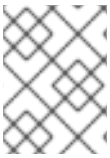
- Optionally register new FunctionModifiers on the start of the ExecutionFactory to handle translation of these functions. Given that the syntax of these functions is same as other typical functions, this probably is not needed - the default translation should work.

- Create a new translator JAR containing your custom ExecutionFactory. Refer to Section 11.1,

"Packaging" and Section 11.2, "Translator Deployment Overview" for instructions on using the JAR file. Once this extended translator is deployed in JBoss Data Virtualization, use "myoracle" as translator name instead of the "oracle" in your VDB's Oracle source configuration.

12.3.2. Defining a Source Supported Function via Teiid Designer

If you are designing your VDB using Teiid Designer, you can define a function on any "source" model, and that function is automatically added as pushdown function when the VDB is deployed. There is no additional need for adding Java code.



NOTE

The function will be visible only for that VDB; whereas, if you extend the translator, the functions can be used by any VDB.

12.3.3. Defining a Source Supported Function Using Dynamic VDBs

If you are using the Dynamic VDB, and defining the metadata using DDL, you can define your source function in the VDB as

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="DDL"><![CDATA[
           CREATE FOREIGN FUNCTION SCORE (val integer) RETURNS integer;
           .... (other tables, procedures etc)
         ]]>
        </metadata>
    </model>
</vdb>
```

By default, in the Dynamic VDBs, metadata for the Source models is automatically retrieved from the source if they were JDBC, File, WebService. The File and WebService sources are static, so one can not add additional metadata on them. However on the JDBC sources you can retrieve the metadata from source and then user can append additional metadata on top of them. For example

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="NATIVE,DDL"><![CDATA[
           CREATE FOREIGN FUNCTION SCORE (val integer) RETURNS integer;
         ]]>
        </metadata>
    </model>
</vdb>
```

The above example uses NATIVE metadata type (NATIVE is the default for source/physical models) first to retrieve schema information from source, then uses DDL metadata type to add additional metadata. Only metadata not available via the NATIVE translator logic would need to be specified via DDL.

Alternatively, if you are using custom MetadataRepository with your VDB, then provide the "function" metadata directly from your implementation. ex.

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="{metadata-repo-module}"></metadata>
    </model>
</vdb>
```

In the above example, user can implement MetadataRepository interface and package the implementation class along with its dependencies in a JBoss EAP module and supply the module name in the above XML. For more information on how to write a Metadata Repository refer to the section on Custom Metadata Repository.

CHAPTER 13. ADMIN API

13.1. ADMIN API

In most circumstances administration will be performed using the Management Console or AdminShell, but it is also possible to invoke administration functionality directly in Java through the Admin API.

All classes for the Admin API are in the client JAR under the **`org.teiid.adminapi`** package.

13.2. CONNECTING

An Admin API 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 JBoss Data Virtualization installation for the appropriate admin port - the default is 9999.

13.3. ADMINISTRATION METHODS

Administration 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 API Documentation for more information.

See Also:

- Section C.2, “Download API Documentation”

CHAPTER 14. CUSTOM LOGGING

14.1. CUSTOMIZED LOGGING

Red Hat JBoss Data Virtualization provides a great deal of information via its logging system. To control logging level, contexts, and log locations, you should be familiar with the server's **`standalone.xml`** or **`domain.xml`** configuration file and the "logging" subsystem. Refer to the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more details about the different contexts available.

If you want a custom log handler, you must write a custom java.util.logging.Handler. To do so, ensure you place the implementation class in the "org.jboss.teiid" module as a jar. Next, define its name, along with any dependencies it may need, in the module.xml file.

14.2. COMMAND LOGGING API

If you want to build a custom appender for command logging that will have access to **`java.util.logging.LogRecord`**s to the "COMMAND_LOG" context, the handler will receive a message that is an instance of **`LogRecord`**. This object will contain a parameter of type **`org.teiid.logging.CommandLogMessage`**. The relevant JBoss Data Virtualization classes are defined in the **`teiid-api-[versionNumber].jar`**. The CommandLogMessage includes information about VDB, session, command SQL, etc. CommandLogMessages are logged at the DEBUG level. An example follows.

```
package org.something;
import java.util.logging.Handler;
import java.util.logging.LogRecord;

public class CommandHandler extends Handler {
    @Override
    public void publish(LogRecord record) {
        CommandLogMessage msg = (CommandLogMessage)record.getParameters()
[0];
        //log to a database, trigger an email, etc.
    }

    @Override
    public void flush() {
    }

    @Override
    public void close() throws SecurityException {
    }
}
```

14.3. AUDIT LOGGING API

If you want to build a custom appender for command logging that will have access to **`java.util.logging.LogRecord`**s to the "AUDIT_LOG" context, the handler will receive a message that is an instance of **`LogRecord`**. This object will contain a parameter of type **`org.teiid.logging.AuditMessage`**. The relevant JBoss Data Virtualization classes are defined in the **`teiid-api-[versionNumber].jar`**. AuditMessages are logged at the DEBUG level. An example follows.

```
package org.something;
import java.util.logging.Handler;
import java.util.logging.LogRecord;

public class AuditHandler extends Handler {
    @Override
    public void publish(LogRecord record) {
        AuditMessage msg = (AuditMessage)record.getParameters()[0];
        //log to a database, trigger an email, etc.
    }

    @Override
    public void flush() {
    }

    @Override
    public void close() throws SecurityException {
    }
}
```

14.4. CONFIGURATION

Now that you have developed a custom handler class, package the implementation in a JAR file, then copy this JAR file into the modules directory and edit the **module.xml** file in the same directory and add

```
    <resource-root path="{your-jar-name}.jar" />
```

then edit **standalone.xml** or **domain.xml** file, locate the "logging" subsystem and add the following entries.

```
    <custom-handler name="COMMAND" class="org.teiid.logging.CommandHandler"
      module="org.jboss.teiid">
    </custom-handler>

    ..other entries

    <logger category="org.teiid.COMMAND_LOG">
        <level name="DEBUG"/>
        <handlers>
            <handler name="COMMAND"/>
        </handlers>
    </logger>
```

Change the above configuration accordingly for AuditHandler, if you are working with Audit Messages.

CHAPTER 15. RUNTIME UPDATES

15.1. DATA UPDATES

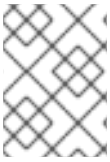
Data change events are used by JBoss Data Virtualization to invalidate resultset cache entries. Resultset cache entires are tracked by the tables that contributed to their results. By default JBoss Data Virtualization will capture internal data events against physical sources and distribute them across the cluster. This approach has a couple of limitations. First, updates are scoped only to their originating VDB/version. Second, updates made outside of JBoss Data Virtualization are not captured. To increase data consistency, external change data capture tools can be used to send events to JBoss Data Virtualization. From within a cluster the **`org.teiid.events.EventDistributorFactory`** and **`org.teiid.events.EventDistributor`** can be used to distribute change events. The **`EventDistributorFactory`** can be looked up by its name "teiid/event-distributor-factory". See the example below.

```
InitialContext ctx = new InitialContext();
EventDistributorFactory edf =
(EventDistributorFactory)ctx.lookup("teiid/event-distributor-factory");
EventDistributor ed = edf.getEventDistributor();
ed.dataModification(vdbName, vdbVersion, schema, tableName);
```

This will distribute a change event for schema.tableName in VDB vdbName.vdbVersion.

When externally capturing all update events, the "detect-change-events" property in the "teiid" subsystem can be set to false, so change events will not be duplicated. By default, this property is set to true.

Use of other **`EventDistributor`** methods to manually distribute other events is not always necessary. See System Procedures in *Red Hat JBoss Development Guide: Reference Material*for SQL based updates.



NOTE

Using the **`org.teiid.events.EventDistributor`** interface you can also update runtime metadata. Refer to the API.

15.2. RUNTIME METADATA UPDATES

Runtime updates via system procedures and DDL statements are by default ephemeral. They are effective across the cluster only for the currently running VDBs. With the next VDB start the values will revert to whatever is stored in the VDB. Updates may be made persistent by configuring an **`org.teiid.metadata.MetadataRepository`**.

An instance of a **`MetadataRepository`** can be installed via the VDB file.

In Designer based VDB, you can edit the vdb.xml file in the META-INF directory or use Dynamic VDB file as below.

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="VIRTUAL">
         <metadata type="{jboss-as-module-name}"></metadata>
    </model>
</vdb>
```

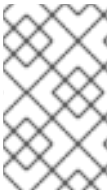
In the above code fragment, replace the {jboss-as-module-name} with a JBoss EAP module name that has library that implements the **`org.teiid.metadata.MetadataRepository`** interface and defines file "META-INF/services/org.teiid.metadata.MetadataRepository" with name of the implementation file.

The **`MetadataRepository`** repository instance may fully implement as many of the methods as needed and return null from any unneeded getter.

NOTE

It is not recommended to directly manipulate **`org.teiid.metadata.AbstractMetadataRecord`** instances. System procedures and DDL statements should be used instead since the effects will be distributed through the cluster and will not introduce inconsistencies.

`org.teiid.metadata.AbstractMetadataRecord` objects passed to the **`MetadataRepository`** have not yet been modified. If the **`MetadataRepository`** cannot persist the update, then a **`RuntimeException`** should be thrown to prevent the update from being applied by the runtime engine.



NOTE

The MetadataRepository can be accessed by multiple threads both during load (if using dynamic VDBs) or at runtime with DDL statements. Your implementation should handle any needed synchronization.

15.3. COSTING UPDATES

See *Red Hat JBoss Data Virtualization Development Guide: Reference Material* for the system procedures **`SYSADMIN.setColumnStats`** and **`SYSADMIN.setTableStats`**. To make costing updates persistent **`MetadataRepository`** implementations should be provided for the following methods:

```
TableStats getTableStats(String vdbName, int vdbVersion, Table table);
void setTableStats(String vdbName, int vdbVersion, Table table, TableStats
tableStats);
ColumnStats getColumnStats(String vdbName, int vdbVersion, Column column);
void setColumnStats(String vdbName, int vdbVersion, Column column,
ColumnStats columnStats);
```

15.4. SCHEMA UPDATES

See *Red Hat JBoss Data Virtualization Development Guide: Reference Material* for supported DDL statements. To make schema updates persistent implementations should be provided for the following methods:

```
String getViewDefinition(String vdbName, int vdbVersion, Table table);
void setViewDefinition(String vdbName, int vdbVersion, Table table, String
viewDefinition);
String getInsteadOfTriggerDefinition(String vdbName, int vdbVersion, Table
table, Table.TriggerEvent triggerOperation);
void setInsteadOfTriggerDefinition(String vdbName, int vdbVersion, Table
table, Table.TriggerEvent triggerOperation, String triggerDefinition);
```

```
boolean isInsteadOfTriggerEnabled(String vdbName, int vdbVersion, Table
table, Table.TriggerEvent triggerOperation);
void setInsteadOfTriggerEnabled(String vdbName, int vdbVersion, Table
table, Table.TriggerEvent triggerOperation, boolean enabled);
String getProcedureDefinition(String vdbName, int vdbVersion, Procedure
procedure);
void setProcedureDefinition(String vdbName, int vdbVersion, Procedure
procedure, String procedureDefinition);
LinkedHashMap<String, String> getProperties(String vdbName, int
vdbVersion, AbstractMetadataRecord record);
void setProperty(String vdbName, int vdbVersion, AbstractMetadataRecord
record, String name, String value);
```

CHAPTER 16. CUSTOM METADATA REPOSITORY

16.1. CUSTOM METADATA REPOSITORY

Traditionally the metadata for a Virtual Database is built by Teiid Designer and supplied to the JBoss Data Virtualization engine through a VDB archive file. This VDB file contains the metadata files called INDEX files, that are then read by a specific instance of MetadataRepository by name INDEX.

In the Dynamic VDB scenario, currently there are three import types available: NATIVE, DDL and FILE.

16.2. NATIVE

This is only applicable on source models (also default). When used, the metadata for the model is retrieved from the source database itself.

Example 16.1. Sample vdb.xml file

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="NATIVE"></metadata>
    </model>
</vdb>
```

NOTE

If a user implements the **getMetadata** method on the **ExecutionFactory** class, NATIVE uses this method to retrieve the metadata from source.

16.3. DDL

Example 16.2. Sample vdb.xml file

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="DDL">
          **DDL Here**
        </metadata>
    </model>
</vdb>
```

This is applicable to both source and view models. When DDL is specified as the metadata import type, the model's metadata can be defined as DDL. See the section about DDL Metadata in *Red Hat JBoss Data Virtualization Development Guide: Reference Material*.

16.4. FILE

Example 16.3. Sample vdb.xml file

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="DDL-FILE">/accounts.ddl</metadata>
    </model>
</vdb>
```

This is applicable to both source and view models in zip VDB deployments. See the section about DDL Metadata in *Red Hat JBoss Data Virtualization Development Guide: Reference Material*

16.5. CUSTOM

If above provided metadata facilities are not sufficient for user's needs then user can extend the MetadataRepository class provided in the org.teiid.api JAR to plug-in their own metadata facilities into the JBoss Data Virtualization engine.

1. Users can write metadata facility that is based on reading data from database or a JCR repository or so forth. Here is an example:

```
package com.something;

import org.teiid.metadata.MetadataRepository;
...

public class CustomMetadataRepository extends MetadataRepository {
    @Override
    public void loadMetadata(MetadataFactory factory,
ExecutionFactory executionFactory, Object connectionFactory)
        throws TranslatorException {
        /* Provide implementation and fill the details in factory */
        ...
    }
}
```

2. Build a JAR archive with above implementation class and create file named org.teiid.metadata.MetadataRepository in the META-INF/services directory with these contents:

```
com.something.CustomMetadataRepository
```

3. Deploy the JAR to Red Hat JBoss EAP as a module under the modules directory. Follow the below steps to create a module.
    - Create a directory called modules/com/something/main.
    - Under this directory create a "module.xml" file that looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.something">
    <resources>
        <resource-root path="something.jar" />
    </resources>
    <dependencies>
        <module name="javax.api"/>
        <module name="javax.resource.api"/>
        <module name="org.jboss.teiid.common-core"/>
        <module name="org.jboss.teiid.teiid-api" />
    </dependencies>
</module>
```

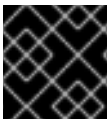
- Copy the jar file under this same directory. Make sure you add any additional dependencies if required by your implementation class under dependencies.
- Restart the server.

This is how you configure the VDB with the custom metadata repository you have created:

Example 16.4. Sample vdb.xml file

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="{metadata-repo-module}"></metadata>
    </model>
</vdb>
```

When the VDB is deployed, it will call the CustomMetadataRepository instance for metadata of the model. Using this you can define metadata for single model or for the whole VDB pragmatically.



IMPORTANT

Be careful about holding state and synchronization in your repository instance.

16.6. USING MULTIPLE IMPORTERS

When you define the metadata import type for a model, you can also define a comma-separated list of importers. By doing so, you will ensure that all of the repository instances defined by import types are consulted in the order in which they have been defined. Here is an example:

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="NATIVE,DDL">
          **DDL Here**
        </metadata>
    </model>
</vdb>
```

In this model, the NATIVE importer is used first, then the DDL importer is used to add additional metadata to the NATIVE-imported metadata.

16.7. DEVELOPMENT CONSIDERATIONS

- **`MetadataRepository`** instances are created on a per VDB basis and may be called concurrently for the load of multiple models.
- See the **`MetadataFactory`** and the **`org.teiid.metadata`** package javadocs for metadata construction methods and objects. For example if you use your own DDL, then call the **`MetadataFactory.parse(Reader)`** method. If you need access to files in a VDB zip deployment, then use the **`MetadataFactory.getVDBResources`** method.
- Use the **`MetadataFactory.addPermission`** and add **`MetadataFactory.addColumnPermission`** method to grant permissions on the given metadata objects to the named roles. The roles should be declared in your vdb.xml, which is also where they are typically tied to container roles.

16.8. PREPARSER

If it is desirable to manipulate incoming queries prior to being handled by Teiid logic, then a custom pre-parser can be installed. Use the PreParser interface provided in the org.teiid.api jar to plug-in a pre-parser for the Teiid engine. See Setting up the build environment to start development.

```
import org.teiid.PreParser;
...

package com.something;

public class CustomPreParser implements PreParser {

    @Override
    public String preParse(String command, CommandContext context) {
        //manipulate the command
    }
}
```

Next, build a JAR archive with above implementation class and create a file named org.teiid.PreParser in the META-INF/services directory with these contents:

```
com.something.CustomPreParser
```

The JAR has now been built. Deploy it in the JBoss AS as a module under jboss-as/modules directory. Now create a module:

Create a directory called jboss-as/modules/com/something/main. In it create a "module.xml" file with these contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.something">
    <resources>
        <resource-root path="something.jar" />
    </resources>
    <dependencies>
```

```
        <module name="javax.api"/>
        <module name="javax.resource.api"/>
        <module name="org.jboss.teiid.common-core"/>
        <module name="org.jboss.teiid.teiid-api" />
    </dependencies>
</module>
```

Copy the jar file under this same directory. Make sure you add any additional dependencies if required by your implementation class under dependencies.

Use the command line interface or modify the configuration to set the preparser-module in the Teiid subsystem configuration to the appropriate module name.

Restart the server

IMPORTANT

Development Considerations Changing the incoming query to a different type of statement is not recommended as are any modifications to the number or types of projected symbols.

APPENDIX A. EXECUTING COMMANDS

A.1. EXECUTION MODES

The JBoss Data Virtualization query engine uses the **ExecutionFactory** class to obtain the **Execution** interface for the command it is executing. The query is sent to the translator as a set of objects. Refer to Section B.1, “Language” for more information.

Translators are allowed to support any subset of the available execution modes.

Table A.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 canceled and closed. ProcedureExecution also extends ResultSetExecution, since procedures may also return resultsets.

A.2. EXECUTIONCONTEXT

The **org.teiid.translator.ExecutionContext** class provides information related to the current execution. An instance of **ExecutionContext** is available for each **Execution**. Various 'get' methods are provided; for example, **ExecutionContext.getRequestIdentifier()** and **ExecutionContext.getSession()** are provided for logging purposes. Specific usage is highlighted in this guide where applicable.

A.3. GENERATED KEYS

To see if the user query expects generated keys to be returned, consult the **CommandContext.isReturnAutoGeneratedKeys()** method. If you wish to return generated keys, you must first create a **GeneratedKeys** instance to hold the keys with the **returnGeneratedKeys** method passing the column names and types of the key columns. Only one **GeneratedKeys** may be associated with the **CommandContext** at any given time.

A.4. SOURCE HINTS

The JBoss Data Virtualization source meta-hint is used to provide hints directly to source executions via user or transformation queries. See the reference for more on source hints. If specified and applicable, the general and source specific hint will be supplied via the ExecutionContext methods

`getGeneralHint` and `getSourceHint`. See the source for the `OracleExecutionFactory` for an example of how this source hint information can be utilized.

A.5. RESULTSETEXECUTION

Typically most commands executed against translators are QueryExpression. While the command is being executed, the translator provides results via the `ResultSetExecution.next()` method. This method returns null to indicate the end of results. Note: the expected batch size can be obtained using the `ExecutionContext.getBatchSize()` method and used as a hint in fetching results from the EIS.

A.6. UPDATE EXECUTION

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

A.7. 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 using the `getOutputParameterValues()` method.

A.8. ASYNCHRONOUS EXECUTIONS

In some scenarios, a translator will execute asynchronously and allow the executing thread to perform other work. To allow this, it is recommended that a `DataNotAvailableException` is thrown during a retrieval method, rather than explicitly waiting or sleeping for the results.

NOTE

The `DataNotAvailableException` should not be thrown by the execute method, as that can result in the execute method being called multiple times. The `DataNotAvailableException` may take a delay parameter or a `Date` in its constructor to indicate when to poll next for results. Any non-negative delay value indicates the time in milliseconds until the next polling should be performed.

The `DataNotAvailableException.NO_POLLING` exception (or any DataNotAvailableException with a negative delay) can be thrown so that processing will resume (via `ExecutionContext.dataAvailable()`).

Since the execution (and the associated connection) is not closed until the work has completed, care must be taken if using asynchronous executions that hold a lot of state.

A positive retry delay is not a guarantee of when the translator will be polled next. If the `DataNotAvailableException` is consumed while the engine thinks more work can be performed or there are other shorter delays issued from other translators, then the plan may be queued again

earlier than expected. You should throw a **DataNotAvailableException** again if your execution is not yet ready. Alternatively the **DataNotAvailableException** may be marked as strict, which does provide a guarantee that the **Execution** will not be called until the delay has expired or the given **Date** has been reached. Using the **Date** constructor makes the **DataNotAvailableException** automatically strict. Due to engine thread pool contention, platform time resolution, etc. a strict **DataNotAvailableException** is not a real-time guarantee of when the next poll for results will occur, only that it will not occur before then.

NOTE

If your **ExecutionFactory** returns only asynch executions that perform minimal work, then consider having **ExecutionFactory.isForkable** return false so that the engine knows not to spawn a separate thread for accessing your **Execution**.

A.9. REUSABLE EXECUTIONS

A translator may return instances of **ReusableExecutions** for the expected Execution objects. There can be one **ReusableExecution** per query executing node in the processing plan. The lifecycle of a **ReusableExecution** is different that a normal **Execution**. After a normal creation/execute/close cycle the **ReusableExecution.reset** is called for the next execution cycle. This may occur indefinitely depending on how many times a processing node executes its query. The behavior of the **close** method is no different from a regular **Execution**, it may not be called until the end of the statement if lobs are detected and any connection associated with the **Execution** will also be closed. When the user command is finished, the **ReusableExecution.dispose()** method will be called.

In general **ReusableExecutions** are most useful for continuous query execution and will also make use of the **ExecutionCotext.dataAvailable()** method for Asynchronous Executions. See *Red Hat JBoss Development Guide: Client Development* for more information about executing continuous statements. In continuous mode the user query will be continuously re-executed. A **ReusableExecution** allows the same **Execution** object to be associated with the processing plan for a given processing node for the lifetime of the user query. This can simplify asynch resource management, such as establishing queue listeners. Returning a null result from the **next()** method **ReusableExecution** as with normal **Executions** indicates that the current pushdown command results have ended. Once the **reset()** method has been called, the next set of results should be returned again terminated with a null result.

See the kit examples for a reusable execution example.

A.10. BULK EXECUTION

Non batched **Insert, Update, Delete** commands may have multi-valued **Parameter** objects if the capabilities shows support for BulkUpdate. Commands with multi-valued Parameters represent multiple executions of the same command with different values. As with BatchedUpdates, bulk operations should be executed atomically if possible.

A.11. 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.

A.12. COMMAND CANCELLATION

Commands submitted to JBoss Data Virtualization 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, JBoss Data Virtualization 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.

APPENDIX B. COMMAND LANGUAGE

B.1. LANGUAGE

JBoss Data Virtualization 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 command sent to the Translator. However, it is possible to specify that your Translator can only accept certain kinds of constructs via the capabilities defined on the **`ExecutionFactory`** class. Refer to the section on 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 JBoss Data Virtualization Javadoc.

B.2. 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. A 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
- **`AggregateFunction`** - represents an aggregate function which holds a single expression
- **`WindowFunction`** - represents a window function which holds an AggregateFunction (which is also used to represent analytical functions) and a WindowSpecification
- **`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 until one of them evaluates to TRUE, then evaluates the associated THEN clause.
- **Array** - represents an array of expressions, currently only used by the engine in multi-attribute dependent joins - see the supportsArrayType capability.

B.3. 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
- **SubqueryComparison** - 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/SIMILAR TO/LIKE_REGEX criteria that compares string values

B.4. 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 the section on command form for more information.

B.5. 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) and a **Limit** (represent a SQL LIMIT clause) or a **With** (represents a SQL WITH clause).

B.6. SELECT STRUCTURE

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

B.7. SETQUERY STRUCTURE

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

B.8. WITH STRUCTURE

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

B.9. 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**.

B.10. 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.

B.11. 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.

B.12. 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.

B.13. BATCHEDUPDATES STRUCTURE

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

B.14. THE TYPE FACILITY

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

The TypeFacility 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 JBoss Data Virtualization 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.

B.15. LANGUAGE MANIPULATION

In Translators that support a richer set of capabilities, there is often a need to manipulate or create language interfaces with a similar syntax to those being translated to. This is often the case when translating to a language comparable to SQL. 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 a **`Condition`** apart based on **`AND`** operators. These utilities are helpful for breaking apart a criteria into individual filters that your translator can implement.

B.16. RUNTIME METADATA

JBoss Data Virtualization uses a library of metadata, known as *runtime metadata* for each virtual database (VDB) that is deployed. The runtime metadata is a subset of the metadata defined by the models contributing to your VDB. While building your VDB in the Designer, you can define what called an *Extension Model*, that defines any number of arbitrary properties on a model and its objects. At runtime, using the runtime metadata interface, you can use properties that were defined at design time to define 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.

B.17. 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.

B.18. 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 B.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();
```

B.19. VISITOR 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 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 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, and so forth) 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.

B.20. PROVIDED VISITORS

The **`SQLStringVisitor`** is a special visitor that can traverse a tree of language interfaces and output the equivalent JBoss Data Virtualization 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.

B.21. 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 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();
```

APPENDIX C. APPENDIX

C.1. TEMPLATE FOR RA.XML

The following is an example of an ra.xml file that can be used 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.

C.2. DOWNLOAD API DOCUMENTATION

Javadocs for JBoss Data Virtualization can be found on the Red Hat Customer Portal .

Procedure C.1. Download API Documentation

1. Open a web browser and navigate to https://access.redhat.com/jbossnetwork.
2. From the **Software Downloads** page, when prompted for a **Product**, select **Data Virtualization**. This will present a table of files to download for the latest version of the product.
3. Change the **Version** to the current release if required.
4. Look for **JBoss Data Virtualization *VERSION* Javadocs** in the table and select **Download**.

C.3. JBOSS DATA VIRTUALIZATION FUNCTIONS AND ORDER OF PRECEDENCE

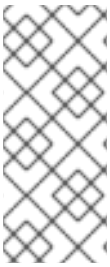
There are three classes of functions in JBoss Data Virtualization:

- System functions (effectively scoped to SYS) and are known at design time.
- Pushdown functions (also effectively scoped to SYS) and are supplied by translators.
- UDFs which are schema scoped (except for legacy function models) and are defined via

metadata.

When resolved, system functions take preference - no schema qualification is necessary. But you can introduce for example a concat UDF and call it as schema.concat(...).

Pushdown functions then take preference.



NOTE

It possible that two translators will declare the same function with the same root name, but JBoss Data Virtualization currently does not treat this as an ambiguity. The primary reason is that in Teiid Designer, pushdown functions must be redeclared in metadata to be used (either with the legacy function model or with source functions). So it is assumed that the first matching definition is correct.

Schema scoped functions are last in preference and require qualification if there are conflicting names.

APPENDIX D. REVISION HISTORY

Revision 6.2.0-54444 **Thu Dec 10 2015** **David Le Sage**

Updates for 6.2.