Informatica

Informatica® Development Platform

Cloud Data Integration Connector Toolkit Developer Guide

Informatica Development Platform Cloud Data Integration Connector Toolkit Developer Guide
November 2022

Publication Date: 2022-11-08

Table of Contents

Preface

Use the Cloud Data Integration Informatica® Connector Toolkit Developer Guide to develop connectors for the Informatica Intelligent Cloud Services℠ platform. Learn how to build a connector by defining the connection attributes, type system, and runtime behavior of the connector. This guide also includes information about how to build the sample MySQL_cloud connector.

Informatica Resources

Informatica provides you with a range of product resources through the Informatica Network and other online portals. Use the resources to get the most from your Informatica products and solutions and to learn from other Informatica users and subject matter experts.

Informatica Network

The Informatica Network is the gateway to many resources, including the Informatica Knowledge Base and Informatica Global Customer Support. To enter the Informatica Network, visit https://network.informatica.com.

As an Informatica Network member, you have the following options:

- Search the Knowledge Base for product resources.
- View product availability information.
- Create and review your support cases.
- Find your local Informatica User Group Network and collaborate with your peers.

Informatica Knowledge Base

Use the Informatica Knowledge Base to find product resources such as how-to articles, best practices, video tutorials, and answers to frequently asked questions.

To search the Knowledge Base, visit https://search.informatica.com. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team at KB_Feedback@informatica.com.

Informatica Documentation

Use the Informatica Documentation Portal to explore an extensive library of documentation for current and recent product releases. To explore the Documentation Portal, visit https://docs.informatica.com.

If you have questions, comments, or ideas about the product documentation, contact the Informatica Documentation team at infa_documentation@informatica.com.

Informatica Product Availability Matrices

Product Availability Matrices (PAMs) indicate the versions of the operating systems, databases, and types of data sources and targets that a product release supports. You can browse the Informatica PAMs at https://network.informatica.com/community/informatica-network/product-availability-matrices.

Informatica Velocity

Informatica Velocity is a collection of tips and best practices developed by Informatica Professional Services and based on real-world experiences from hundreds of data management projects. Informatica Velocity represents the collective knowledge of Informatica consultants who work with organizations around the world to plan, develop, deploy, and maintain successful data management solutions.

You can find Informatica Velocity resources at http://velocity.informatica.com. If you have questions, comments, or ideas about Informatica Velocity, contact Informatica Professional Services at ips@informatica.com.

Informatica Marketplace

The Informatica Marketplace is a forum where you can find solutions that extend and enhance your Informatica implementations. Leverage any of the hundreds of solutions from Informatica developers and partners on the Marketplace to improve your productivity and speed up time to implementation on your projects. You can find the Informatica Marketplace at https://marketplace.informatica.com.

Informatica Global Customer Support

You can contact a Global Support Center by telephone or through the Informatica Network.

To find your local Informatica Global Customer Support telephone number, visit the Informatica website at the following link:
https://www.informatica.com/services-and-training/customer-success-services/contact-us.html.

To find online support resources on the Informatica Network, visit https://network.informatica.com and select the eSupport option.

CHAPTER 1

Introduction to the Informatica Connector Toolkit

This chapter includes the following topics:

Informatica Connector Toolkit Overview

Use the Informatica Connector Toolkit to build a connector that provides connectivity between a data source and Informatica Intelligent Cloud Services.

Although Informatica supports generic ODBC connectivity and allows access to any data source that has a standards-compliant ODBC driver, building a connector by using the Informatica Connector Toolkit offers several advantages. In cases where no ODBC driver is available, building a connector by using the Informatica Connector Toolkit might be the only solution.

When you use the Informatica Connector Toolkit to create a connector, you can build functionality related to the data source. You can preserve data type integrity and metadata lineage of the data source when you retain the type system of the data source and perform optimal data type conversions.

The Informatica Connector Toolkit consists of libraries, plug-ins, and sample code to assist you in developing connectors for Informatica Intelligent Cloud Services. You can use the Informatica Connector perspective in the Eclipse IDE to quickly develop a connector in an Eclipse environment.

The Informatica Connector Toolkit simplifies the following processes:

- Development. You can use the wizards in the Informatica Connector perspective to rapidly develop a connector. The wizards simplify the use of internal components and dependencies when you develop a connector.
- Testing. After you define the connector components, you can test the connection, metadata, and run-time components of the connector.
- Deployment. You can deploy the connector on Informatica Intelligent Cloud Services.

The Informatica Connector Toolkit API is written in a combination of Java. The connection definition and metadata definition are available in Java. The run-time interfaces are available in Java.

Supported Features

The following table lists the features that Informatica Connector Toolkit supports for Informatica Intelligent Cloud Services:

Connector Component	Features	Informatica Intelligent Cloud Services
Connection	Connection configuration	Yes
	Connection pooling	Yes
Metadata	Bulk processing	Yes
	Custom query capability	Yes
	Filter operation	Yes
	Join operation	Yes
	Lookup	No
	Mapping in advanced mode	Yes
	Metadata import configuration	No
	Metadata write capability	Yes
	Multiple endpoint metadata objects Previously known as native metadata object.	No
	Object path override capability	Yes
	Partitioning capability	Yes **Note:** You can only implement static partitioning.
	Pushdown capability	Yes
	Select operation	No
	Sort operation	No
	SQL Transformation	Yes
	Stored procedures	Yes
Runtime	C Runtime	No
	Java Runtime **Note:** Java runtime supports Commit, Rollback, IsEqual, and IsTransactional APIs for Informatica Intelligent Cloud Services.	Yes

Informatica Connector Perspective

The views in the Informatica Connector perspective enable you to develop connectors for Informatica Intelligent Cloud Services.

After you click the **Create New Connector** icon in the Eclipse Workbench toolbar, you can choose to switch to the Informatica Connector perspective. You can also open the Informatica Connector perspective from the **Window** menu in the Eclipse IDE.

The Informatica Connector perspective consists of the Connector Navigator view and Connector Progress view in the Eclipse Workbench window, and icons in the Eclipse toolbar. You can use the Connector Navigator and Connector Progress views to define, edit, or test the connector components and view the progress of the connector project. Use the **Create New Connector** icon in the Toolbar to create a connector project. You can use the **View or Create Messages** icon in the Eclipse toolbar to view or create messages for the connector components.

Connector Navigator

You can use the Connector Navigator view to add, define, edit, or test connector components, such as connection, metadata, and run time.

After you create an Informatica Connector project, you can right-click folders in the Connector Navigator view to define, edit, or test connector components. You can also right-click the connector source files and edit them in the Eclipse Workbench editor or any other editor.

Connector Progress

You can use the Connector Progress view to view the completeness of the connector project and to define or edit the connector project components in each phase.

The Connector Progress view consists of the following phases:

Connectivity

You can define, edit, or test the connection component of a connector in the Connectivity phase.

Metadata

You can define and edit data types and endpoint metadata object components of a connector in the Metadata phase. You can also define common metadata for connectors in the Metadata phase. You can test the connector metadata components in the Metadata phase.

Runtime

You can define, edit, and test the connector run-time components in the Runtime phase.

Export/Publish

You can publish the connector to a location or deploy the connector on Informatica Intelligent Cloud Services in the Export/Publish phase.

CHAPTER 2

Installing and Upgrading the Informatica Connector Toolkit

This chapter includes the following topics:

Installing the Informatica Connector Toolkit Overview

Install the Informatica Connector Toolkit and set up the Informatica Connector Toolkit Eclipse plug-in. To set up the Informatica Connector Toolkit Eclipse plug-in, you install the required software on the machine where you plan to develop the connector.

The Informatica Connector Toolkit is part of the Informatica Development Platform (IDP). You can use the Informatica Connector Toolkit installer to install the Informatica Connector Toolkit on the machine in which you plan to develop a connector.

You can get the Informatica Connector Toolkit installer from the following sources:

- Informatica electronic software download site.
  When you purchase an Informatica product and choose to download the software, you receive a site link, user ID, and password to access the Informatica electronic software download site. Follow the instructions on the download site to download the Informatica Connector Toolkit installation file.
- Informatica Technology Network.
  If you are a registered user of the Informatica Technology Network, you can download the Informatica Connector Toolkit installation file from the Informatica Development Platform page. When you download the file, the Informatica Development Network provides you with a password. Use this password when you extract the files from the download file.

When you run the Informatica Connector Toolkit installer, the installer installs the Informatica Connector Toolkit Eclipse plug-in in the Eclipse IDE.

Before You Begin

Before you develop a connector with the Informatica Connector Toolkit, you need to install the required software and analyze the data source.

Run the Informatica Connector Toolkit installer to install the Informatica Connector Toolkit on the machine where you plan to develop the connector. The Informatica Connector Toolkit contains the binaries, tools, samples, and documents that you require to build a connector.

Download the Welcome Kit

The Informatica Partner Program provides a complete set of enablement, marketing, and sales resources and information so that partners can develop and promote their services and solutions in conjunction with Informatica.

Click the following link to download the Partner Welcome KIT:

Download

Install Software

Install the following software on the machine where you plan to develop the connector:

- Eclipse IDE for Java EE Developers with support for Plug-in Development Environment (PDE). Informatica Connector Toolkit supports the following Eclipse versions:

Eclipse	Version	Java
eclipse-juno	SR1	1.7 and above
eclipse-juno	SR2	1.7 and above
eclipse-kepler	SR1	1.7 and above
eclipse-kepler	SR2	1.7 and above
eclipse-luna	SR1	1.7 and above
eclipse-luna	SR2	1.7 and above
eclipse-mars	Milestone 1 (4.5.0M1)	1.7 and above
eclipse-mars	SR1-SR2	1.7 and above
eclipse-neon	M5	1.8 and above
eclipse-oxygen	3a	1.8 and above
eclipse-2020-06	4.16.0	1.8 and above

  Note: For AIX, install Eclipse 4.2.2 or above.

- Azul OpenJDK version 1.8.0_275 or previous sub versions.
- MySQL_Cloud Connector/J JDBC driver version 8.0.13 or later if you use the MySQL_Cloud sample connector.

- YouTube Data API client library for Java version 3 or later if you use the YouTube sample connector.

Configure Environment Variables for OpenJDK on Linux

After you download the Azul OpenJDK version 1.8.0_275 or any previous sub version on a Linux machine, you must configure the environment variables.

Perform the following steps to configure the environment variables:

1. After you download the Azul OpenJDK, extract the .tar file.
2. Set the variable name to JAVA_HOME and the variable value to `<Java installation directory>/jdk`:
   `export JDK_HOME=<Java installation directory>/jdk`
3. Delete the JRE_HOME environment variable.
4. Edit the PATH variable and add `JAVA_HOME/bin` to the variable value.
   `export PATH=$JDK_HOME/bin:$PATH`
5. In the Informatica Connector Eclipse IDE, open **Windows** > **Preferences**.
6. Navigate to **Java** > **Installed JREs**.
7. Click **Add**.
8. Select **Standard VM** as the **JRE Type**.
9. Click **Next**.
10. In the **JRE home** property, specify the `<Java installation directory>/jdk` or click **Directory** to navigate to the `<Java installation directory>/jdk`.
11. Enter a name for the **JRE name** property.
12. Click **Finish**.
13. Click **Apply** and then click **OK**.
14. Restart the Informatica Connector Eclipse IDE.

Configure Environment Variables for Azul OpenJDK on Windows

After you download the Azul OpenJDK version 1.8.0_275 or any previous sub version on a Windows machine, you must configure the environment variables.

Perform the following steps to configure the environment variables:

1. After you download the Azul OpenJDK, extract the .zip file.
2. Open the **Advanced System Properties** from the Windows Control Panel.
3. Click **Environment Variables**.
4. Under System variables, click New.
5. Set the variable name to JAVA_HOME and the variable value to `<Java installation directory>/jdk`.
6. Edit the Path variable and add `JAVA_HOME/bin` to the variable value.
7. In the Informatica Connector Eclipse IDE, open **Windows** > **Preferences**.
8. Navigate to **Java** > **Installed JREs**.
9. Click **Add**.
10. Select **Standard VM** as the **JRE Type**.
11. Click **Next**.

12. In the **JRE home** property, specify the `<Java installation directory>` or click **Directory** to navigate to the `<Java installation directory>`.
13. Enter a name for the **JRE name** property.
14. Click **Finish**.
15. Click **Apply** and then click **OK**.
16. Restart the Informatica Connector Eclipse IDE.

Download the Cloud Data Integration Services Connector Toolkit

Download the Cloud Data Integration Connector Toolkit based on your system requirement:

Operating System	URL
Windows 64-bit	Download
Linux 64-bit	Download

Installation of Eclipse IDE

You can install Eclipse IDE on the machine in which you plan to develop the connector. Use the Eclipse IDE package to install Eclipse IDE.

You must install Eclipse version supported by Informatica Connector Toolkit for the Informatica Connector Toolkit plug-in to work with the Eclipse IDE.

You can download the Eclipse IDE package from the following location:

http://www.eclipse.org/downloads/

For more information about the supported Eclipse versions, see "Install Software" on page 13.

Installing Informatica Connector Toolkit on Windows

Install the Informatica Connector Toolkit to install the Informatica Connector Toolkit Eclipse plug-in and other components that you require to build an connector.

1. Close all other applications.
2. Run the `install.bat` file from the root directory.
3. In the **Welcome** page, click **Next**.

   The **Installation Directory** page appears.
4. Select the installation directory in which you want to install the Informatica Connector Toolkit.
5. Select the installation directory in which you installed Eclipse IDE.

6. Select **Install Apache log4j** to install the log4j plug-in version that Informatica Connector Toolkit requires. The installer installs the required Apache log4j plug-in by default. If you have log4j plug-in installed and do not want to install a different version of the library, clear the **Install Apache log4j** option.
7. Select **Install Apache Commons library** to install the commons library that Informatica Connector Toolkit requires.
8. Click **Next**.

   The **Pre-Installation Summary** page appears.
9. Click **Install**.

   The **Installing** page appears and displays the installation progress.
10. Click **Done** to complete the installation procedure and then exit the installer.

Installed Informatica Connector Toolkit Components

After you install the Informatica SDKs, you can find the Informatica Connector Toolkit in the installation directory where you installed the Informatica Connector Toolkit.

The Informatica Connector Toolkit installation includes the following components to assist you in developing a connector for the Informatica platform:

- Informatica Connector Toolkit API files. Library files of the Informatica Connector Toolkit API.
- Informatica Connector Toolkit Eclipse plug-in. You can use the wizards and menus that the Informatica Connector Toolkit plug-in adds to the Eclipse IDE to develop, test, and deploy a connector.
- Sample MySQL_Cloud connector. The sample MySQL_Cloud connector include source code that you can use as a model to build a connector.

  Note: The sample MySQL_Cloud connector are for illustration purposes only.
- Informatica Connector Toolkit API Reference. Online documentation for the Informatica Connector Toolkit API specification.

Sample Connectors Source Codes

The Informatica Connector Toolkit installation includes the following sample connector source codes:

- Instagram
- MySQL
- MySQL_Cloud
- Redis
- YouTube

CHAPTER 3

Building a Connector

This chapter includes the following topics:

Building a Connector for Cloud Data Integration

The Informatica Connector Toolkit consists of the libraries, plug-ins, and sample code to assist you in developing a connector for the Informatica platform.

To build a connector for Cloud Data Integration, use the Informatica Connector Toolkit and perform the following tasks:

1. Create a connector project and define the basic properties of the connector such as name, ID, and vendor name.
2. Define the connection attributes to connect to the data source. Implement the methods to open connection or close connection to the data source, validate connection, and specify attribute dependencies. Before you define the type system for the connector, you can test and debug the connection components of the connector.
   You can also use connection pooling to reuse the connections instead of creating a new connection each time and optimize the performance.
3. Define the type system for the connector. Specify the data types supported by the data source and the corresponding data types supported by Informatica.
4. Define the connector metadata, create endpoint metadata object, operations for the endpoint metadata object, and partition methods for the operations. Implement the methods to fetch metadata from the data source. You can also test and debug the metadata components of the connector.
   Enable read and write capabilities for mappings on the advanced cluster

   If you want enable read and write capabilities for a mapping that runs on the advanced cluster, complete the following prerequisites:

   1. Click the following link to download and install the Scala binaries:
      https://www.scala-lang.org/download/2.12.8.html
   2. Add the SCALA_HOME environment variable under system variables in the agent machine.
      Set the value of the variable to the installation directory where you installed the Scala binaries.
5. Define the connector run-time behavior that defines how the connector reads from and writes to the data source. Before you deploy the connector, you can test and debug the read capability and write capability of the connector.
6. Deploy the connector to the Cloud Data Integration service.

Step 1. Select the Product Type and Define the Connector Properties

The connector properties describe and identify the connector in the Informatica Intelligent Cloud Services Administrator. Use the Informatica Connector Toolkit to create an connector project and define the connector properties.

1. From the Eclipse IDE, click the **Create New Connector** button ().
   The **Create Informatica Connector Project** dialog box appears.
2. Select **Data Integration** as the product type to select Cloud Data Integration.
3. Click **Next**.
   The **Create Informatica Connector Project** dialog box appears.
4. Enter the following connector project details:

Property	Description
Connector ID	An identifier that uniquely identifies the connector.
Connector Name	Name of the connector. The connector name is an alphanumeric string. The first character of the name must be a letter.
Vendor Name	Name of the vendor building the connector.
Vendor ID	A unique identifier for the vendor.
Version	The version number of the connector must have the format x.x.x and must include numeric characters. For example: 1.0.0.
Description	Description of the connector.

5. Click **Finish**.
   The connector project appears in the **Connector Navigator** view of the Eclipse IDE.

Step 2. Define the Connection Attributes

The connection attributes of a connector determine how the connector connects to the data source. Use the Informatica Connector Toolkit to define the connection attributes and specify the libraries required to connect to the data source.

1. In the **Connector Navigator** view, right-click the project and select **Add Connection**.
   The **Add Connection** dialog box appears.
2. Enter the connection type name.
3. Select **Add New** and enter the category name.
   The category name that you configure corresponds to the connection object category.
4. Click **Next**.
   The **Connection Attributes** page appears.
5. Click **Add** to enter each connection attribute.
   The attributes that you configure correspond to the connection object properties. The **Add Attribute** dialog box appears.

6. Enter the following properties for the connection attribute:

Attribute Property	Description
Name	Name of the connection attribute.
Display Name	Display name for the connection attribute.
Description	Description of the connection attribute.
Data Type	Data type of the connection attribute.
Default Value	Default value for the attribute.
Min Length	Minimum length for the value of the attribute.
Max Length	Maximum length for the value of the attribute.
Encrypted	Indicates whether you can encrypt the attribute.
Mandatory	Indicates whether a connection attribute requires a value. If you set the Mandatory property to true but you do not display the attribute on the connection management user interface, you must set a default value for the attribute.
Hidden	Indicates whether you can hide the attribute.
Has Dependent Fields	Indicates whether the attribute has dependent fields.
Allowed Values	List of values allowed for the attribute.

7. Click **OK**.
8. Click **Next**.
   The **Configure Connection Pages** page appears.
   a. Group the connection attributes under one or more connection sections.
   b. To create a new connection section, click **Add Section** and enter the section name, section title, tool tip, and section description.
   c. To create a new connection page, click **Add Page** and enter the page name and page description.
   d. To change the order in which connection sections appear in a connection page, use **Move Up** or **Move Down**.
9. Click **Next**.
   The **Connection Pooling** page appears.
10. Set the connection pool through the Informatica Connector Toolkit interface or API.
    - To use connection pooling through the API, select the **Support Connection Pooling through API** check box. Values for the connection pooling properties are set through the code.

- To use connection pooling through the Informatica Connector Toolkit interface, select the **Supports Connection Pooling** check box. Then, enter the details for the following connection pooling properties:

Property	Description
Supports Custom Compare	Enables a custom comparison of connection objects. The isEqual() API takes the connection object as an input argument and compares it with the current calling object. The API isEqual() returns true if both the objects are similar. Else, it returns false. The connector developer overrides the isEqual() method to provide a custom comparison of connection objects.
Maximum Idle Connections per key	Determines the maximum number of connection handles that can remain idle in the connection pool. If there are one or more idle instances available in the sub-pool associated with a given key, an idle instance is selected based on the LIFO principle and returned.
Maximum Total Connections in the pool	Determines the maximum number of objects that can be pooled for all distinct keys.
Maximum Total Connections per key	Determines the maximum number of objects that can be pooled for a particular key.
Minimum evictable idle time in milliseconds	Determines the time in milliseconds that makes the object eligible to get destroyed when the time for the object in an idle state elapses.
Maximum wait time in milliseconds	Maximum wait time for a particular request to get an object from the pool, before it returns failure.
Abandoned time out in milliseconds	Prevents an abandoned connection object from staying in the memory. If a connection object that is borrowed from the pool is not returned to the pool within the specified abandoned time out, that object is considered abandoned and then destroyed by the pool.
Time between eviction run in milliseconds	Dictates the interval between successive eviction of idle objects from the pool.
Test on create	Tests the validity of objects before adding them to the pool. Select the check box to test the objects for validity before adding them to the pool.
Test on borrow	Tests the validity of objects before borrowing them from the pool. Select the check box to test the objects for validity before borrowing them from the pool.

Property	Description
Test on return	Tests the borrowed objects for validity before returning them to the pool. Select the check box to test the borrowed objects for validity before returning them to the pool.
Number of retries for invalid connections	The number of times to retry the metadata operation on receiving an `InvalidConnectionHandleException` from the connector. For each retry attempt, the Secure Agent picks a fresh connection object from the pool. Default value is 3.

11. Click **Next**.
    The **Configure Libraries** page appears.
12. Click **Add** to select each library that the connector requires to connect to the data source.
13. Click **Generate Code**.
    After you define the connection attributes, the Informatica Connector Toolkit generates the following Java files:

    <ConnectorName>ConnectInfoAdapter.java

    <ConnectorName>Connection.java
14. Update the ConnectInfoAdapter.java file to implement connection validation and attribute dependancies. Also, update the Connection.java file to implement the methods that open and close connection to the data source.
15. You can test and debug the connection to the data source.

Note: If you regenerate code for the connection project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica perspective. You have to manually edit the source code and make changes if you add, remove, or modify connection attributes.

Step 3. Define the Type System

Categorize each data type in the data source into one of the Cloud Data Integration data types or Java data types supported by the Informatica Connector Toolkit API. Use the Informatica Connector Toolkit to create a type system or generate and use a predefined type system.

Defining the Type System Manually

You can manually define a type system that maps the data types that the data source supports with the data types that Cloud Data Integration supports.

1. In the **Connector Navigator** view, right-click the project and select **Define Typesystem**.
   The **Define Type System** dialog box appears.
2. To manually define a type system that matches the data types in the data source, select **Manually Create Type System**.
3. To map the native data types to the Cloud Data Integration data types, select **Native to Platform Type**.
4. To map the native data types to the Java data types, select **Native to Java Type**.
5. Click **Add** to configure each type system attribute.
   The **Add Type** dialog box appears.

6. Enter the following properties for the type system attribute:

Property	Value
Type Name	Name for the native type system attribute.
Comments	Comments for the native type system attribute.
Best Platform Type for Read/ Best Java Type for Read	The platform or Java data type that maps best to the native data type when the connector reads from the data source.
Properties	Based on the selected best platform or Java type for read, you must set one or more of the following native type properties: - Precision properties such as maximum precision and default precision. By default, the maximum precision is displayed for each data type. - Scale properties such as maximum scale, default scale, and minimum scale. - Length properties such as maximum length and default length. The maximum length of any attribute cannot be greater than Integer.MAX_VALUE, which is 2,147,483,647. - Unit of length such as characters, bytes, and bits. - Date properties such as hour, minute, second, year, month, day, and time zone.

7. If the platform data type or Java data type is the best match when writing to the native data type, select **Mapped** and then select **Best Native Type to Write**.
   For example, to map the Java INTEGER data type to the MySQL data type INT as one of the Best Native Type to Write, select Mapped in the INTEGER row and then select Best Native Type to Write.
8. To map a platform data type or Java data type to the native data type but not as a best match, select only **Mapped**.
   For example, if the Java SHORT data type maps to the MySQL data type INT but might result in loss of data when converted, select Mapped in the INTEGER row.
9. Click **OK**.
   The **Define Type System** dialog box appears.
10. After you add and map the required native data types for the connector, click **Generate Code**.

Generating a Predefined Type System

You can generate a type system with the basic data types that Cloud Data Integration supports. Use the basic data types in the predefined type system for use with REST-based procedures.

1. In the **Connector Navigator** view, right-click the project and select **Define Typesystem**.
   The **Define Type System** dialog box appears.
2. To generate and use basic data types that the Cloud Data Integration supports, select **Use Predefined Type System**.
3. To map the native data types to the Cloud Data Integration data types, select **Native to Platform Type**.
4. To map the native data types to the Java data types, select **Native to Java Type**.
5. Click **Generate Code**.

Step 4. Define the Common Metadata

You can specify additional metadata details, such as schema name and foreign key name for data sources in which data is stored in a schema. The connector uses the specified schema name and foreign key name during the read operation to retrieve data.

Note: By default, the **Supports Schema** option is enabled for all connectors. Disable the **Supports Schema** option if your connector does not support schema, and click **Save**.

1. In the **Connector Navigator** view, right-click the project and select **Edit Common Metadata**.

   The **Edit Common Metadata** dialog box appears.
2. If the connector supports schema, select **Supports Schema** and add the **Schema Display Name**.
3. If the connector supports foreign key, select **Supports Foreign Key** and add the **Foreign Key Display Name**.
4. Click **Generate Code**.

Step 5. Define the Connector Metadata

The connector metadata represents the metadata in the data source for which you build the connector. You can define metadata definitions to represent the differently structured metadata objects of the data source.

Use the Informatica Connector Toolkit to define the connector metadata. You can represent the metadata for data sources in which data is stored as records and for procedures in data sources. You can manually create native metadata for procedures or use swagger specifications to define the native metadata.

You can use the procedure pattern to define endpoint metadata objects for Informatica Intelligent Cloud Services connectors.

Define the following connector components to specify the connector metadata:

- Endpoint metadata definition for the connector. You can define multiple native metadata definitions for a connector. For example, you can create different endpoint metadata objects for tables, views, and synonyms in a relational data source.
- Record extensions and field extensions. You can define record extensions and field extensions to define additional metadata for records and fields.
- Read and write capability for the connector. You can add attributes that you can use to read from or write to the data source.
- Pushdown capability for the connector. You can add attributes that you can use to push as much of the transformation logic as possible to the target database.

  Note: Before you enable pushdown capability for the connector, you must copy the `com.infa.products.expr.jexpr.jar` file from

  `<ICT installation directory>/CCI/plugins/infa` to `<ICT installation directory>/ICT/Application/plugins` and restart the Informatica Connector Toolkit.
- Import dialog box settings. You can define import options that appear in Cloud Data Integration when a connector consumer imports a data object.

Note: If you regenerate code for the endpoint metadata definition project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica Connector perspective. You have to manually edit the source code and make changes if you add, remove, or change the endpoint metadata attributes.

Defining the Connector Metadata for Record Pattern

For data sources in which data is stored as records, you can define the endpoint metadata definition for the connector by using the record pattern type.

1. In the Connector Navigator view, right-click the project and select **Add Endpoint Metadata Definition**.
   The **Add Endpoint Metadata Definition** dialog box appears.
2. Enter the endpoint metadata details.
   The following table describes the properties to enter:

Property	Description
Name	Name for the endpoint metadata.
Display Name	Display name for the endpoint metadata.
Description	Description of the endpoint metadata.

3. Specify the pattern type as record to create a endpoint metadata object based on the record structure of the data source.
4. To specify a custom query to import a data source, select **Custom Query**.
   When you select **Custom Query**, you can select **Query** as the source type to import a data source.

   Note: Custom Query capability is supported only for Informatica Intelligent Cloud Services. Custom query is not applicable for Cloud Toolkit (CTK) connectors.
5. To validate the value of the parameter of the data source object, select **Object Path Override**.

   Note: The Object Path Override capability is supported only for Informatica Intelligent Cloud Services. Object Path Override is not applicable for CTK connectors.
6. Click **Next**.
7. To add additional metadata information for records, select **Add Record Extension** and add the following properties for the attribute:

Attribute Property	Description
Name	Name of the attribute.
Display Name	Display name for the connection attribute.
Description	Description of the connection attribute.
Data Type	Data type of the connection attribute.
Default Value	Default value for the attribute.
Min Length	Minimum length for the value of the attribute.
Max Length	Maximum length for the value of the attribute. The maximum length of any attribute cannot be greater than Integer.MAX_VALUE, which is 2,147,483,647.
Encrypted	Indicates whether you can encrypt the attribute.

Attribute Property	Description
Mandatory	Indicates whether a connection attribute requires a value. If you set the Mandatory property to True but you do not display the attribute on the connection management user interface, you must set a default value for the attribute.
Hidden	Indicates whether you can hide the attribute.
Allowed Values	List of values allowed for the attribute.

8. Click **Next**.
   The **Endpoint Metadata Field** page appears.
9. To add additional metadata information for fields, select **Add Field Extension** and add the attribute properties.
10. Click **Next**.
    The **Read Capability** page appears.
11. To define read capability for the endpoint metadata object, select **Enable Read Capability**.
12. Select whether the connector supports lookup of data when the connector reads from the data source.
13. Select **Enable read in advanced mode** to enable read capability for mappings in advanced mode.
    For more information, see Chapter 9, “Mappings in advanced mode” on page 68.
14. Select whether the connector supports join and filter operations when the connector reads from the data source.
    - To specify operators and expression syntax recognized by the Cloud Data Integration for the join or filter operation, select **Platform Expression**.

      Note: If you plan to add key range partitioning capability for the connector, you must select support for filter operation and platform expression.
    - To specify an expression for the join or filter operation that is specific to the data source for which you build the connector, select **Native Expression**.

    Note: When a connector supports platform filter, you can use the default operators such as `=`, `!=`, `>`, `>=`, `<`, and `<=` for filter conditions. You can specify advanced operators such as `Contains`, `Starts With`, `Ends With`, `Is Null`, and `Is Not Null` for filters when you define the read capability for the endpoint metadata object.

    For the code changes to include the advanced operators, see the RuntimeDataAdapter class of MySQL_Cloud sample connector in the following location:

    `source\ict\samples\MySQL_Cloud`
15. Select whether the connector supports sort to retrieve data from the data source in a specific order.
16. To add read capability attributes, click **Add** and add the attribute properties.

You can also define the following parameterization and partitioning support for an attribute:

- When you specify the read attribute properties, you can define whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. For example, you can parameterize the user name and add user names to a configuration file and then run the same mapping to read data of different users.

  You can define the following parameterization support for an attribute:
  Full Parameterization

  The attribute supports parameterization. You can parameterize the value of an attribute completely.

  Partial Parameterization

  The attribute supports partial parameterization. You can parameterize a part of the attribute value.

  No

  The attribute does not support parameterization.
- You can select **Override Partitions** to specify if the attribute can be overridden for each partition. Implement the <ConnectorID><NMOName>AutoPartitioningMetadataAdapter file to define the partition support.

You can also define whether the attribute supports parameterization and if the attribute can be overridden for each partition.

17. Click **Next**.
The **Write Capability** page appears.
18. To define write capability for the endpoint metadata object, select **Enable Write Capability** and add the attribute properties.
When you specify the write attribute properties, you can define whether you can parameterize the attribute.

    You can define the following parameterization support for an attribute:
    Full Parameterization

    The attribute supports parameterization. You can parameterize the value of an attribute completely.

    Partial Parameterization

    The attribute supports partial parameterization. You can parameterize a part of the attribute value.

    No

    The attribute does not support parameterization.
19. Select whether the connector supports upsert operation when the connector writes to the target.
When you select **Enable 'Upsert' support**, Informatica Connector Toolkit adds the `UpdateMode` attribute to the write capability attribute list.

    You can specify one of the following values for the `UpdateMode` attribute during runtime:

    - **Update As Update**. If you specify the **Update As Update** value, you must implement the upsert logic so that the connector updates an existing row while writing to the target.
    - **Update Else Insert**. If you specify the **Update Else Insert** value, you must implement the upsert logic so that the connector updates an existing row if the row exists in the target, else inserts a row while writing to the target.
20. Select whether the connector supports bulk processing to write large amounts of data to the target.
When you select **Enable Write Bulk API**, specify the property `-DENABLE_WRITER_BULK_PROCESSING=true` in the Secure Agent properties.

Note: Bulk processing capability is supported only for Informatica Intelligent Cloud Services. Bulk processing is not applicable for CTK connectors.

21. Select **Enable write in advanced mode** to enable write capability for mappings in advanced mode. For more information, see Chapter 9, “Mappings in advanced mode” on page 68.
22. Click **Next**.
    The **Pushdown Capability for the Native Metadata** page appears.
23. To define pushdown capability for the endpoint metadata object using the native connection, select **Enable Pushdown Capability** and add the attribute properties.
    When you specify the pushdown attribute properties, you can define the adapter ID for the source. Define whether the connector supports source or full pushdown or pushdown to single or multiple targets. When you enable pushdown capability, you must add at least one source adapter ID.

    Note: Pushdown capability is supported only for Informatica Intelligent Cloud Services. Pushdown capability is not applicable for CTK connectors.
24. Select the transformations that the connector must support for pushdown optimization. You can select from the following transformations:
    - Filter
    - Joiner
    - Sorter
    - Expression
    - Aggregator
    - Lookup
    - Router
    - Union
    - Update Strategy
    - Sequence Generator
25. Click **Next**.
    The **Native Metadata Partitioning Capability** page appears.
26. Select whether the connector supports partitioning capability for the read operation.
27. To configure the connector to fetch partition information from the data source, select the **Dynamic** partitioning method and implement the partition logic. Extend the AutoPartitioningMetadataAdapter class to implement the partition logic.
28. To configure the connector to get partition information from the user, select the **Static** partitioning method. The user enters the partition information, such as number of partitions or key range.
    - If you want to implement a partition logic based on the partition information that the user specifies, select **Fixed**. Implement the partition logic in the DataAdapter class. The user can specify the required partition information before the connector reads from the data source.
    - If the tables in the data source support key range partitioning, select **Key Range**. The user can specify the partition keys and key range type before the connector reads from the data source. If you add support for key range partitioning, ensure that the connector supports filter operation and platform expression. You need not implement the partition logic for key range partitioning because the Informatica Connector Toolkit implements key range partitioning as a filter query.

      Note: Key range partitioning is supported only for Big Data Management.
29. Select whether the connector supports partitioning capability for the write operation. By default, the dynamic partitioning method is selected for partition-enabled write operations. Extend the AutoPartitioningMetadataAdapter class to implement the partition logic.

30. Click **Next**.
    The **Import Dialog Box Settings** page appears.
31. In the **Metadata Import Dialog Box Settings** section, select the metadata import options that appear in the client tool when a connector consumer creates a data object and click **Save**.
32. If you are developing a connector for the Informatica Intelligent Cloud Services, specify whether the connector supports DDL generation and select the supported modifications to the data source schema in the **Metadata Write Settings** section.
    You can configure the connector to support the following modifications in the data source schema:
    Supports DDL

    You can choose how Data Integration handles changes that you make to the data object schemas. To refresh the schema every time the mapping task runs, you can enable dynamic schema handling in the task. You can select DDL support to create, update, and delete metadata in an external system using the specified objects and options.

    Create Object

    To add the capability to create objects in the target, select **Create Object**. To add the capability to drop the existing object and then create a new object, select **Drop and Create**.

    Alter Object

    To add the capability to alter objects in the target, select **Alter Object**. To add the capability to create a object if the object to alter does not exist, select **Alter Else Create**.

    Drop Object

    To add the capability to drop objects in the target, select **Drop Object**.
33. Click **Generate Code**.
    After you define the connector metadata, the Informatica Connector Toolkit generates the `<NMOName>MetadataAdapter.java` file in the **Metadata** folder. Implement the following methods in the `<NMOName>MetadataAdapter.java` file to import metadata.

    populateObjectCatalog()

    Populates metadata details in the import wizard for the connector consumer.

    Note: Data preview in Cloud Data Integration does not work if the values of the Record.setName and Record.setNativeName methods are different.

    populateObjectDetails()

    Gets metadata from the data source based on the import dialog options settings.

If you configured metadata write settings for the connector, implement the `writeObjects` method in the `<NMOName>MetadataAdapter.java` file.

Note: If you regenerate code for the endpoint metadata definition project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica perspective. You have to manually edit the source code and make changes if you add, remove, or change the endpoint metadata attributes.

Define the Connector Metadata for Procedure Pattern

For procedures and functions in data sources, you can define the endpoint metadata definition for the Informatica Intelligent Cloud Services connectors by using the procedure pattern type.

You can use the manual method to create the endpoint metadata object for the procedure or use a REST-based specification. You can choose to use an existing swagger specification or generate and use a swagger specification.

Defining the Connector Metadata for Procedure Pattern Manually

When you use the procedure pattern type to define the endpoint metadata for the Connector, you can use the manual method to define the endpoint metadata object.

1. In the Connector Navigator view, right-click the project and select **Add Endpoint Metadata Definition**.
   The **Add Endpoint Metadata Definition** dialog box appears.
2. Enter the endpoint metadata details.
   The following table describes the properties to enter:

Property	Description
Name	Name for the endpoint metadata.
Display Name	Display name for the endpoint metadata.
Description	Description of the endpoint metadata.

3. Specify the pattern type as procedure to create a endpoint metadata object for procedures or functions in a data source.
4. To manually define the endpoint metadata object for the data source with procedure pattern type, select **Manual** as the creation method.
5. Click **Next**.
   The **Endpoint Metadata Procedure Definition** page appears.
6. To add additional metadata information for procedure, select **Add Procedure Extension** and add the following properties for the attribute:

Attribute Property	Description
Name	Name of the attribute.
Display Name	Display name for the connection attribute.
Description	Description of the connection attribute.
Data Type	Data type of the connection attribute.
Default Value	Default value for the attribute.
Min Length	Minimum length for the value of the attribute.
Max Length	Maximum length for the value of the attribute. The maximum length of any attribute cannot be greater than Integer.MAX_VALUE, which is 2,147,483,647.
Encrypted	Indicates whether you can encrypt the attribute.
Mandatory	Indicates whether a connection attribute requires a value. If you set the Mandatory property to True but you do not display the attribute on the connection management user interface, you must set a default value for the attribute.

Attribute Property	Description
Hidden	Indicates whether you can hide the attribute.
Allowed Values	List of values allowed for the attribute.

7. Click **Next**.
   The **Endpoint Metadata Parameter Definition** page appears.
8. To add additional metadata information for parameters, select **Add Parameter Extension** and add the attribute properties.
9. Click **Next**.
   The **Endpoint Metadata Call Capability Attributes** page appears.
10. To add call capability attributes, click **Add** and add the attribute properties.
    You can select **Override Partitions** to specify if the attribute can be overridden for each partition. Implement the <ConnectorID><NMOName>AutoPartitioningMetadataAdapter file to define the partition support.
11. Click **Next**.
    The **Import Dialog Box Settings** page appears.
12. In the **Metadata Import Dialog Box Settings** section, select the metadata import options that appear in the client tool when a connector consumer creates a data object and click **Save**.
13. Click **Generate Code**.
    After you define the connector metadata, the Informatica Connector Toolkit generates the `<NMOName>MetadataAdapter.java` file in the **Metadata** folder. Implement the following methods in the `<NMOName>MetadataAdapter.java` file to import metadata.

    populateObjectCatalog()

    Populates metadata details in the import wizard for the connector consumer.

    Note: Data preview in the PowerCenter client or Informatica Intelligent Cloud Services does not work if the values of the Record.setName and Record.setNativeName methods are different.

    populateObjectDetails()

    Gets metadata from the data source based on the import dialog options settings.

If you configured metadata write settings for the connector, implement the `writeObjects` method in the `<NMOName>MetadataAdapter.java` file.

Note: If you regenerate code for the endpoint metadata definition project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica Connector perspective. You have to manually edit the source code and make changes if you add, remove, or change the endpoint metadata attributes.

Defining the Connector Metadata for Procedure Pattern by Using REST-based Specification

You can define the endpoint metadata definition for the connector by using an existing REST-based specification or generate and use a REST-based specification.

1. In the Connector Navigator view, right-click the project and select **Add Endpoint Metadata Definition**.
   The **Add Endpoint Metadata Definition** dialog box appears.
2. Enter the endpoint metadata details.

The following table describes the properties to enter:

Property	Description
Name	Name for the endpoint metadata.
Display Name	Display name for the endpoint metadata.
Description	Description of the endpoint metadata.

3. Specify the pattern type as procedure to create a endpoint metadata object for procedures or functions in a data source.
4. To create endpoint metadata object based on a REST-based specification, select REST-based creation method.
5. To use an existing swagger specification, click **Browse** and open the JSON file. After you open the JSON file, skip to step 7.
6. To generate a swagger specification by sampling, select **Generate swagger specification by sampling** and then click **Next**.
   Perform the following steps to sample the REST end point:
   a. Enter the base URL.
   b. Click **Add** to get the procedure details.
   c. Specify the **Path**, **Display Name**, **Method**, **Content Type**, and **Accept Type** properties.
   d. Add the parameter details.
   e. Click **Test** to validate and sample the REST end point.
   f. Click **OK** to add the procedure details.
7. Click **Generate Code**.
   After you define the connector metadata, the Informatica Connector Toolkit generates the `<NMOName>MetadataAdapter.java` file in the **Metadata** folder. To implement features specific to the data source, you can also modify code in the following methods in the `<NMOName>MetadataAdapter.java` file to import metadata.

   populateObjectCatalog()

   Populates metadata details in the import wizard for the connector consumer.

   Note: Data preview in Cloud Data Integration does not work if the values of the Record.setName and Record.setNativeName methods are different.

   populateObjectDetails()

   Gets metadata from the data source based on the import dialog options settings.

If you configured metadata write settings for the connector, implement the `writeObjects` method in the `<NMOName>MetadataAdapter.java` file.

Note: If you regenerate code for the endpoint metadata definition project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica perspective. You have to manually edit the source code and make changes if you add, remove, or change the endpoint metadata attributes.

Defining the Connector Metadata for Procedure Pattern by Using SQL Transformation

When you use the procedure pattern type to define the endpoint metadata for the connector, you can use an SQL transformation to define the endpoint metadata object.

1. In the Connector Navigator view, right-click the project and select **Add Endpoint Metadata Definition**.
   The **Add Endpoint Metadata Definition** dialog box appears.
2. Enter the endpoint metadata details.
   The following table describes the properties:

Property	Description
Name	Name for the endpoint metadata.
Display Name	Display name for the endpoint metadata.
Description	Description of the endpoint metadata.
Use Existing Runtime	Determines if the endpoint metadata object uses an existing runtime or a new runtime.
Endpoint Metadata Object	Displays a list of endpoint metadata objects for which you implemented runtime. If you want to use the existing runtime, you can select one of the endpoint metadata objects from the list. **Note:** The list is displayed only when you select **Use Existing Runtime** option.

3. Specify the pattern type as procedure to create a endpoint metadata object for procedures or functions in a data source.
4. To define the endpoint metadata object for the data source with procedure pattern type by using SQL transformation, select SQL Transformation as the creation method.
5. Click **Next**.
   The **Endpoint Metadata Procedure Definition** page appears.
6. To add additional metadata information for the procedure pattern type, select **Add Procedure Extension** and add the following properties for the attribute:

Attribute Property	Description
Name	Name of the attribute.
Display Name	Display name for the connection attribute.
Description	Description of the connection attribute.
Data Type	Data type of the connection attribute.
Default Value	Default value for the attribute.
Min Length	Minimum length for the value of the attribute.

Attribute Property	Description
Max Length	Maximum length for the value of the attribute. The maximum length of any attribute cannot be greater than Integer.MAX_VALUE, which is 2,147,483,647.
Encrypted	Indicates whether you can encrypt the attribute.
Mandatory	Indicates whether a connection attribute requires a value. If you set the Mandatory property to True but you do not display the attribute on the connection management user interface, you must set a default value for the attribute.
Hidden	Indicates whether you can hide the attribute.
Allowed Values	List of values allowed for the attribute.

7. Click **Next**.
   The **Endpoint Metadata Parameter Definition** page appears.
8. To add additional metadata information for parameters, select **Add Parameter Extension** and add the attribute properties.
9. Click **Next**.
   The **Endpoint Metadata Call Capability Attributes** page appears.
10. To add call capability attributes, click **Add** and add the attribute properties.
11. Click **Next**.
    The **Import Dialog Box Settings** page appears.
12. In the **Metadata Import Dialog Box Settings** section, select the metadata import options that appear in the client tool when you create a data object, and click **Save**.
13. Click **Generate Code**.

    Note: No code changes are required in metadata connector. For more information, see "Code Changes for SQL Transformation" on page 85.

Defining the Connector Metadata for Procedure Pattern by Using Stored Procedures

A stored procedure is a prepared SQL code that you can save. Hence, the code can be reused multiple times. Connectors can define the procedure-based endpoint metadata object by using the stored procedure specification.

When you use the procedure pattern type to define the endpoint metadata for the connector, you can use a stored procedure to define the endpoint metadata object

1. In the Connector Navigator view, right-click the project and select **Add Endpoint Metadata Definition**.
   The **Add Endpoint Metadata Definition** dialog box appears.
2. Enter the endpoint metadata details.

The following table describes the properties:

Property	Description
Name	Name for the endpoint metadata.
Display Name	Display name for the endpoint metadata.
Description	Description of the endpoint metadata.
Use Existing Runtime	Determines if the endpoint metadata object uses an existing runtime or a new runtime.
Endpoint Metadata Object	Displays a list of endpoint metadata objects for which runtime is implemented. If you want to use the existing runtime, you can select one of the endpoint metadata objects from the list. **Note:** The list is displayed only when you select the **Use Existing Runtime** option.

3. Specify the pattern type as procedure to create a endpoint metadata object for procedures or functions in a data source.
4. To define the endpoint metadata object for the data source with procedure pattern type by using SQL transformation, select SQL Transformation as the creation method.
5. Click **Next**.
   The **Endpoint Metadata Procedure Definition** page appears.
6. To add additional metadata information for the procedure pattern type, select **Add Procedure Extension** and add the following properties for the attribute:

Attribute Property	Description
Name	Name of the attribute.
Display Name	Display name for the connection attribute.
Description	Description of the connection attribute.
Data Type	Data type of the connection attribute.
Default Value	Default value for the attribute.
Min Length	Minimum length for the value of the attribute.
Max Length	Maximum length for the value of the attribute. The maximum length of any attribute cannot be greater than Integer.MAX_VALUE, which is 2,147,483,647.
Encrypted	Indicates whether you can encrypt the attribute.
Mandatory	Indicates whether a connection attribute requires a value. If you set the Mandatory property to True but you do not display the attribute on the connection management user interface, you must set a default value for the attribute.

Attribute Property	Description
Hidden	Indicates whether you can hide the attribute.
Allowed Values	List of values allowed for the attribute.

7. Click **Next**.
   The **Endpoint Metadata Parameter Definition** page appears.
8. To add additional metadata information for parameters, select **Add Parameter Extension** and add the attribute properties.
9. Click **Next**.
   The **Endpoint Metadata Call Capability Attributes** page appears.
10. To add call capability attributes, click **Add** and add the attribute properties.
11. Click **Next**.
    The **Import Dialog Box Settings** page appears.
12. In the **Metadata Import Dialog Box Settings** section, select the metadata import options that appear in the client tool when you create a data object, and click **Save**.
13. Click **Generate Code**.
    After you define the adapter metadata, the Informatica Connector Toolkit generates the MetadataAdapter.java file in the Metadata folder.

    Implement the following methods in the MetadataAdapter.java file to import metadata:

    - populateObjectCatalog(). Populates metadata details in the import wizard for the connector consumer.
    - populateObjectDetails(). Gets metadata from the data source based on the import dialog options settings.

    Note: If you regenerate the code for the endpoint metadata definition project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica perspective. You have to manually edit the source code and make changes if you add, remove, or change the endpoint metadata attributes.

    For more information, see "Code Changes for Stored Procedures" on page 91.

Package the resource or configuration files

You can package the resource or configuration files as part of the project jar.

Create a resources folder inside the project, and place the resource files.

For example, If you want to add a configuration file `custom_config.xml` for the metadata.adapter project, create a resources folder and place the configuration file in the following location:

```
<eclipse workspace>/<project_name>/usr/metadata.adapter/src
```

When you build, test, and export the connector, the configuration file is included as part of the metadata adapter jar.

Enable metadata logger

You can enable the logger to log messages in the metadata phase.

The following sample code shows the message string passed to the info() method of the Logger class:

```
logger.info("Metadata Phase in progress");
```

The messages are logged in the tomcat logs in the following directory of the agent machine:

```
<ICT installation directory>/apps/Data_Integration_Server/logs/tomcat
```

Test Metadata from the Data Source

After you define the endpoint metadata objects, you can test metadata that you import from the data source.

To debug when you test metadata from the data source, use the same debug configuration that you used to test the connection to the data source. You can also set breakpoints in the code that you want to debug.

After you define the debug configuration, you can launch the **Test Metadata** dialog box to test the connection definition and test the metadata import from the data source.

1. In the **Connector Progress** view, select Test<NMOName> under the **Test Metadata** section.

   The **Test Metadata** dialog box appears with the connection attributes that you defined for the connector.
2. Enter values for the connection attributes to connect to the data source.
3. Click **Connect**.

   The connector retrieves the metadata from the data source and the metadata appears in the **Test Metadata** page.
4. Browse and verify the retrieved metadata.
5. Click **Close**.

Step 6. Implement the Connector Run-time Behavior

Use the Informatica Connector Toolkit to implement the connector run-time behavior in C/C++ or Java. The connector run-time behavior defines how the connector performs operations, such as to establish a connection, close a connection, prepare SQL statements, and run SQL statements.

You can set up the run-time implementation for each endpoint metadata object of the connector. For example, if you define an endpoint metadata object with read capability and another endpoint metadata object with write capability, you can choose to implement the run time for the first endpoint metadata object with read capability in C++ and implement the run time for the other endpoint metadata object with write capability in Java.

You can also define the connector run-time behavior to support pre and post commands to perform tasks before and after a mapping run. For example, you can define the connector run-time behavior to support a pre command that initializes environment variables before the mapping run.

Implement the Connector Run-Time Behavior in Java

You can use the **Run-Time Implementation** wizard in the Informatica Connector Toolkit to implement the run-time behavior in Java.

1. In the **Connector Navigator** view, right-click the project and select **Runtime** > **<NMOName>** > **Set Up**.

   The **Run-Time Implementation** wizard appears.
2. Choose to implement the run-time behavior in Java.
3. Add the required run-time library files and generate the `<ConnectorID><NMOName>DataAdapter.java` and `<ConnectorID>DataConnection.java` files.

4. Implement the following methods in the `<ConnectorID><NMOName>DataAdapter.java` file:
   - initDataSourceOperation. Implement this method to perform tasks before the mapping runs. For example, you can implement code to initialize environment variables. The scope of the RuntimeConfig and Metadata handles available in this method is within the initDataSourceOperation method.
   - deinitDataSourceOperation. Implement this method to perform tasks after the mapping runs. The scope of the RuntimeConfig and Metadata handles available in this method is within the deinitDataSourceOperation method.
   - initDataSourceOperation
   - deinitDataSourceOperation
   - initDataSession
   - deinitDataSession
   - read
   - write
   - reset (Optional. Implement this method if the connector supports the lookup operation.)
   - beginDataSession
   - endDataSession
5. Implement the following methods in the `<ConnectorID>DataConnection.java` file:
   - connect
   - disconnect
6. To support data preview in the Informatica Intelligent Cloud Services, implement the following method in the `<ConnectorID>ASOOperationObjMgr.java` file:

   prepareRuntimeOperation

Create Messages

You can use the Informatica Connector Toolkit to create, edit, or delete messages and handle exceptions that occur during the design time or run time of the connector.

When you create messages, you specify the message text and message code, and you can include information on the message severity, cause, and user action. After you create messages, you can implement the code to handle exceptions. When you implement the code to handle the exception, you pass the message as an argument to the exception handing method.

Create design-time messages to handle design-time exceptions, such as service exceptions. Create run-time messages to handle run-time exceptions.

To create messages, perform the following steps:

1. In the **Connector Navigator** view, right-click the project and select **View and Create Messages**. The **Messages** dialog box appears.
2. Click **Add**.
   The **Add New Message** dialog box appears.
3. Enter an ID for the message.
4. Enter a code for the message. At run time, the message code and the message text appears in the session log.
5. Specify the severity of the message.

6. Enter the message text.
   You can include parameters in the message text and specify the parameters in Java message format.

   The following example shows parameters used in Java message format:

   Connection User [{0}], Port [{1,number,integer}], Connection time [{1,number}] milliseconds
7. Enter a description for the message.
8. Enter the cause of the error message.
9. Enter the suggested user action when the user encounters the error.
10. Specify whether the message is a design-time message or a run-time message.
11. Click **OK**.
    The **Messages** dialog box appears.
12. Click **Finish**.
    The message XML files appear under the **Message** folder in the **Connector Navigator** view.

Implement Design-Time Messages

You can use the methods in the com.informatica.sdk.exceptions.ExceptionManager class to implement design-time messages.

To enable localization of messages, implement the createNlsAdapterSDKException() method in the ExceptionManager class.

The following sample code shows the message parameters passed to the createNlsAdapterSDKException() method:

```
ExceptionManager.createNlsAdapterSDKException(ExceptionManager.createNlsAdapterSDKExcepti
on(MessageBundle.getInstance(),
Messages.Test_CONN_SUCC_200, "admin", 5040, 138.76);
```

To implement design-time messages that do not require localization, implement the createNonNlsAdapterSDKException() method in the ExceptionManager class.

The following sample code shows the message string passed to the createNonNlsAdapterSDKException() method:

```
ExceptionManager.createNonNlsAdapterSDKException("Unknown error:" + e.getMessage());
```

Implement Run-Time Messages

To implement run-time messages that require localization, implement the logMessage() helper method in the <ConnectorID><NMOName>DataAdapter.java file. The logMessage() method logs messages to the session log.

The following Java sample code shows the message parameters passed to the logMessage() method:

```
logMessage(Messages.Test_CONN_SUCC_200, "admin", 5040, 138.76);
```

The following C sample code shows the message parameters passed to the INFAADPLogMessage() method:

```
INFAADPLogMessage(infaDataSessionHandle, INFA_MSG_ERROR,INFA_TRACE_NONE,
CONN_ID,"admin",12,12.3);
```

To implement the run-time messages that do not require localization, implement the logger.logMessage() method.

Step 7. Test the Read and Write Capabilities of the Connector

Before you deploy the connector in the Informatica domain, you can test the read and write capabilities of the connector.

After you define the connector run-time components, you can use the **Test Read** and **Test Write** wizards to test the read and write capabilities of the connector. After you debug and fix issues in the read and write capabilities of the connector, you can deploy the connector in the Informatica domain.

You can test the read and write capabilities of the connector only for the Windows platform.

Test the Read Capability of the Connector

When you test the read capability of the connector, you test the connection definition, metadata of the data source, and operations that the connector supports. After you specify the test settings and run the test, you can view the result of the read operation, read operation statistics, and the log file. You can test the read capability of the connector only for the Windows platform.

To debug the code, use the same debug configuration that you used to test the connection and metadata components of the connector. You can also set breakpoints in the code that you want to debug.

After you define the debug configuration, you can launch the **Test Read** dialog box to test the read capabilities of the connector.

1. In the **Connector Progress** view, select the endpoint metadata object that appears under **Test Read**.

   The **Test Read** dialog box appears with the default JVM environment settings and tracing level. The Informatica Connector Toolkit uses the JVM settings to run the debug configuration.
2. If required, edit the JVM environment settings. Ensure that you use the same port number that appears in JVM settings for the connection properties in the debug configuration.
3. Select the required tracing level. The default is normal. Based on the amount of detail that you require in the log file, you can override the default tracing level.

   You can set the following types of tracing level:

   None

   Does not override the default tracing level.

   Terse

   Logs initialization information and error messages and notification of rejected data.

   Normal

   Logs initialization and status information, errors encountered, and skipped rows due to transformation row errors. Summarizes mapping results, but not at the level of individual rows. Default is normal.

   Verbose Initialization

   In addition to normal tracing, logs additional initialization details, names of index and data files used, and detailed statistics.

   Verbose Data

   In addition to verbose initialization tracing, logs each row. You can also get detailed statistics on where string data was truncated to fit the precision of a column.
4. Click **Next**.

   The connection attributes that you defined for the connector appears.
5. Enter values for the connection attributes to test the connection to the data source.

6. Click **Connect**.

   The **Test Metadata** page appears with the metadata imported from the data source.

7. Select the endpoint metadata objects and the corresponding endpoint metadata fields to test the read operation.
8. Click **Next**.

   The **Filter Condition** page appears.

9. Select the **Configure Filter** option in the **Filter Condition** page and specify the filter condition.
   - To specify an Informatica platform expression for the filter operation, perform the following steps:
     1. In the **Definition** section, click **Add** to add an Informatica platform expression.
     2. In the **Field** column, select the field to use in the expression.
     3. In the **Operation** column, select a conditional operator to use in the expression.
     4. In the **Value** column, enter a value for the conditional expression.
   - To specify a native expression for the filter operation, enter the expression in the **Definition** section.
10. After you specify expressions for the endpoint metadata object, click **Next**.

    The **Read Capability** page appears.

11. Specify values for the read capability attributes, and then click **Run**.

    The **Result** page appears. You can view the result of the read operation, read operation statistics, and the log file in the **Result** page.

12. Click **Close**.

Test the Write Capability of the Connector

When you test the write capability of the connector, you test the components of the connector and write sample data to the data source. After you specify the test settings and run the test, you can view the result of the write operation, write operation statistics, and the log file.

To debug the code, use the same debug configuration that you used to test the connection and metadata components of the connector. You can also set breakpoints in the code that you want to debug.

After you define the debug configuration, you can launch the **Test Write** dialog box to test the write capabilities of the connector.

1. In the **Connector Progress** view, select the endpoint metadata object that appears under **Test Write**.

   The **Test Write** dialog box appears with the default JVM environment settings and tracing level. The Informatica Connector Toolkit uses the JVM settings to run the debug configuration.

2. If required, edit the JVM environment settings. Ensure that you use the same port number that appears in JVM settings for the connection properties in the debug configuration.
3. Select the required tracing level. The default is normal. Based on the amount of detail that you require in the log file, you can override the default tracing level.

   You can set the following types of tracing level:

   None

   Does not override the default tracing level.

   Terse

   Logs initialization information and error messages and notification of rejected data.

Normal

Logs initialization and status information, errors encountered, and skipped rows due to transformation row errors. Summarizes mapping results, but not at the level of individual rows. This is the default tracing level.

Verbose Initialization

In addition to normal tracing, logs additional initialization details, names of index and data files used, and detailed statistics.

Verbose Data

In addition to verbose initialization tracing, logs each row. You can also get detailed statistics on where string data was truncated to fit the precision of a column.

4. Click **Next**.

   The connection attributes that you defined for the connector appears.
5. Enter values for the connection attributes to test the connection to the data source.
6. Click **Connect**.

   The **Test Metadata** page appears with the metadata imported from the data source.
7. Select an endpoint metadata object to test the write operation.

   The metadata of the endpoint metadata object along with the data type, scale, and precision appears in the **Test Write** page.
8. Select the columns to which you want to write data.
9. Click **Next**.

   The **Test Data** page appears.
10. In the Test Data page, you can load test data from a file or you can generate test data.
    - To load the test data from a file, perform the following steps:
      1. Select the **Load from a File** option. You must load a comma-delimited TXT file or CSV file.

         Note: The date and time data types in the file must have the following timestamp format: MM/DD/YYYY hh24:mm:ss
      2. Click **Browse** and select the file that contains the test data.
    - To generate test data, perform the following steps:
      1. Select the **Auto generate data** option.
      2. Enter the number of rows to generate. You can specify a maximum of 1000 rows.
      3. Click **Generate**. The test data appears in the **Data Preview** section.
      4. If required, you can edit the test data that appears in the **Data Preview** section.
11. Select an insert, update, or delete operation that you want to perform on the target object.

    To perform an update or delete operation, the target object must contain a primary key. If you auto-generate the data, edit the value of the primary key column in the preview section to match with a record in the target object.
12. After you load a test data file or generate test data, click **Next**.

    The **Write Capability** page appears.
13. Specify values for the write capability attributes and then click **Run**.

    The **Result** page appears. You can view the result of the write operation, write operation statistics, and the log file in the **Result** page.
14. Click **Close**.

Step 8. Publish the Connector

Use the Informatica Connector Toolkit to publish the connector. You can publish the connector files to a location.

1. In the **Connector Navigator** view, right-click the project, and select **Publish**.
   The **Publish Informatica Connector** dialog box appears.
2. Click **Browse** and specify a location to publish the connector.
   The **Publish Location** defaults to the connector project workspace.
3. To publish the connector for Cloud Data Integration service, specify a plugin ID and connector version.
   If the connector package already exists, the following warning is displayed:

   ```
   Connector package with same version exists in the publish location. Click on override connector package to proceed or change the connector version.
   ```
4. Select **Override Connector Package** to override the existing package or change the connector version.
5. Click **Next** to view the contents of the client and server bundles.
6. Click **Finish**.

When you publish the connector, the Informatica Connector Toolkit bundles connector artifacts and generates the connector package that you can use for Cloud Data Integration.

If you encounter any error or fail to export the connector, refer to the `<connector_name>_codebuilder.log` file for more information. The `<connector_name>_codebuilder.log` file is available in the `<Eclipse workspace>/<connector project>` folder.

When you publish the connector, the Informatica Connector Toolkit restarts the Secure Agent. The Secure Agent can connect to the data source and you can create connections to the data source in Cloud Data Integration service. If you fail to publish the connector, install the connector manually.

Naming Convention

A naming convention makes it easy to identify the files that belong to a connector.

Use the following guidelines when you name the connector files:

- Determine a name for the connector. Use the connector name as a prefix for the connector file names. The connector name is an alphanumeric string that can include the uppercase and lowercase letters A to Z and the numbers 0 to 9. The first character of the name must be a letter.
- Determine a unique name to identify the company building the connector. The company name is included in the package name for the connector classes.
- Use the connector name when you create a directory for the connector in the Informatica directory:

  `INFA_HOME\plugins\dynamic\`*`ConnectorName`*

The following table lists the recommended naming convention for connector files:

Component	Naming Format
Package for the connector definition and type system classes	com.vendorname.connectorname.connection
Connector definition class	ConnectorNameDefn

Component	Naming Format
Type system class	ConnectorNameTypeSystem
Resource file for the connection management user interface	ConnectorNameBundle.properties or ConnectorNameBundle_lang.properties

Step 8. Test the Connector

After you create and export a new connector, you can test the connector using the automation framework. The framework generates and runs the test cases.

After you export a connector, perform the following tests:

- Code Acceptance Test (CAT) or Unit Test.
  Units tests are standalone tests. You can generate the test case after you upload data in the project explorer. After the test run, a status report is generated without any external dependency.
- Product Acceptance Test (PAT) or Integration Test.
  Integration tests require the configuration details to generate and execute the test cases. It requires user intervention for manual upload of data to the pod.

When the automation framework runs, it performs the following tasks:

1. Analyzes the connector package and generates the test suite. You can use the test suite to test the functionalities in a connector. For example, you can test the connection, verify the record and field details, search records, and test the read, write, filter operations.
2. Runs the test cases to validate the unit and integration tests. You can run the test cases separately.
3. Populates a single unified test report for both the unit and integration tests. You can see the PASS and FAIL status in the report.

Building a Connector for Data Loader

The Informatica Connector Toolkit consists of the libraries, plug-ins, and sample code to assist you in developing a connector for a data loader task.

You can build a connector to connect to a data source and define how the connector reads from the data source. Also, you can only use data sources in which data is stored as records.

If you built a connector for Cloud Data Integration using Informatica Connector Toolkit and need to use the connector in a data loader task, contact the Informatica administrator.

To build a connector for a data loader task, use the Informatica Connector Toolkit and perform the following tasks:

1. Create a connector project and select the product type. Then, select the creation method and define the basic properties of the connector such as name, ID, and vendor name.
2. Define the connection attributes to connect to the data source. Implement the methods to open or close the connection to the data source, validate the connection, and specify the attribute dependencies. Before you define the type system for the connector, you can test and debug the connection components of the connector.

3. Define the type system for the connector. Specify the data types supported by the data source and the corresponding data types supported by Informatica.
4. Define the connector metadata, create endpoint metadata object, operations for the endpoint metadata object, and partition methods for the operations. Implement the methods to fetch metadata from the data source. You can also test and debug the metadata components of the connector.
5. Define the connector run-time behavior that defines how the connector reads from the data source. Before you deploy the connector, you can test and debug the read capability of the connector.
6. Deploy the connector to the data loader task.

Step 1. Select the Product Type and Define the Connector Properties

The connector properties describe and identify the connector in the data loader task.

Use the Informatica Connector Toolkit to create a connector project and define the connector properties.

1. From the Eclipse IDE, click the **Create New Connector** button ().
   The **Create Informatica Connector Project** dialog box appears.
2. To create a connector for a data loader task, select **Data Loader** as the product type.
3. Select one of the following creation methods to create a connector:
   - **JDBC**. Create a JDBC driver-based connector.
   - **Client Libraries**. Create a connector using a third-party library SDK.
   - **Applications**. Create a connector for REST API-based application.
4. Select the authentication type for the Applications creation method. Default is no authentication.
   - **Basic**. Uses user name and password for authentication.
   - **Token**.Uses token-based authentication.
   - **OAuth 1.0**. Uses OAuth 1.0 for authentication.
   - **OAuth 2.0**. Uses OAuth 2.0 for authentication.
   - **API Key**. Uses API key for authentication.
5. Click **Next**.
   The **Create Informatica Connector Project** dialog box appears.
6. Enter the following connector project details:

Property	Description
Connector ID	An identifier that uniquely identifies the connector.
Connector Name	Name of the connector. The connector name is an alphanumeric string. The first character of the name must be a letter.
Vendor Name	Name of the vendor building the connector.
Vendor ID	A unique identifier for the vendor.

Property	Description
Version	The version number of the connector must have the format x.x.x and must include numeric characters. For example: 1.0.0.
Description	Description of the connector.

7. Click **Finish**.
   The connector project appears in the **Connector Navigator** view of the Eclipse IDE.

Step 2. Define the Connection Attributes

The connection attributes of a connector determine how the connector connects to the data source. Use the Informatica Connector Toolkit to define the connection attributes and specify the libraries required to connect to the data source.

1. In the **Connector Navigator** view, right-click the project and select **Add Connection**.
2. In the **Connection Attributes** page, specify the connection attributes based on the creation method.
   - For the **JDBC** creation method, attributes that are common among the JDBC connectors are populated. You can add, edit, or delete the attributes.
   - For the **Client Libraries** creation method, attributes are not populated and you need to configure the attributes.
   - For the **Applications** creation method, if you do not select any authentication type, attributes are not populated and you need to configure the attributes.
     If you select any authentication type, attributes that are common among the Application connectors are populated. You can add, edit, or delete the attributes.
3. Click **Next**.
   The **Configure Connection Pages** page appears.
   a. Group the connection attributes under one or more connection sections.
   b. To create a new connection section, click **Add Section** and enter the section name, section title, tool tip, and section description.
   c. To create a new connection page, click **Add Page** and enter the page name and page description.
   d. To change the order in which the connection sections appear in a connection page, use **Move Up** or **Move Down**.
4. Click **Next**.
   The **Configure Libraries** page appears.
5. Click **Add** to select each library that the connector requires to connect to the data source.
6. Click **Generate Code**.
   After you define the connection attributes, the Informatica Connector Toolkit generates the following Java files:

   ```
   <ConnectorName>ConnectInfoAdapter.java
   ```

   ```
   <ConnectorName>Connection.java
   ```

7. Update the `ConnectInfoAdapter.java` file to implement connection validation and attribute dependencies.
   Also, update the `Connection.java` file to implement the methods that open and close the connection to the data source.
8. You can test and debug the connection to the data source.

Note: If you regenerate code for the connection project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica perspective. You have to manually edit the source code and make changes if you add, remove, or modify connection attributes.

Step 3. Define the Type System

Categorize each data type in the data source into one of the data loader task data types or Java data types supported by the Informatica Connector Toolkit API. Use the Informatica Connector Toolkit to create a type system or generate and use a predefined type system.

Defining the Type System Manually

You can manually define a type system that maps the data types that the data source supports with the data types that the data loader task supports.

1. In the **Connector Navigator** view, right-click the project and select **Define Typesystem**.
   The **Define Type System** dialog box appears.
2. To manually define a type system that matches the data types in the data source, select **Manually Create Type System**.
3. To map the native data types to the data loader task data types, select **Native to Platform Type**.
4. To map the native data types to the Java data types, select **Native to Java Type**.
5. Click **Add** to configure each type system attribute.
   The **Add Type** dialog box appears.
6. Enter the following properties for the type system attribute:

Property	Value
Type Name	Name for the native type system attribute.
Comments	Comments for the native type system attribute.
Best Platform Type for Read/ Best Java Type for Read	The platform or Java data type that maps best to the native data type when the connector reads from the data source.
Properties	Based on the selected best platform or Java type for read, you must set one or more of the following native type properties: - Precision properties such as maximum precision and default precision. By default, the maximum precision is displayed for each data type. - Scale properties such as maximum scale, default scale, and minimum scale. - Length properties such as maximum length and default length. The maximum length of any attribute cannot be greater than Integer.MAX_VALUE, which is 2,147,483,647. - Unit of length such as characters, bytes, and bits. - Date properties such as hour, minute, second, year, month, day, and time zone.

7. If the platform data type or Java data type is the best match when writing to the native data type, select **Mapped** and then select **Best Native Type to Write**.
   For example, to map the Java INTEGER data type to the MySQL data type INT as one of the Best Native Type to Write, select Mapped in the INTEGER row and then select Best Native Type to Write.
8. To map a platform data type or Java data type to the native data type but not as a best match, select only **Mapped**.

For example, if the Java SHORT data type maps to the MySQL data type INT but might result in loss of data when converted, select Mapped in the INTEGER row.

9. Click **OK**.
   The **Define Type System** dialog box appears.
10. After you add and map the required native data types for the connector, click **Generate Code**.

Generating a Predefined Type System

You can generate a type system with the basic data types that the data loader task supports. Use the basic data types in the predefined type system for use with REST-based procedures.

1. In the **Connector Navigator** view, right-click the project and select **Define Typesystem**.
   The **Define Type System** dialog box appears.
2. To generate and use basic data types that the data loader task supports, select **Use Predefined Type System**.
3. To map the native data types to the data loader task data types, select **Native to Platform Type**.
4. To map the native data types to the Java data types, select **Native to Java Type**.
5. Click **Generate Code**.

Step 4. Defining the Connector Metadata

The connector metadata represents the metadata in the data source for which you build the connector. You can define metadata definitions to represent the differently structured metadata objects of the data source. You can represent the metadata for data sources in which data is stored as records.

1. In the Connector Navigator view, right-click the project and select **Add Endpoint Metadata Definition**.
   The **Read Capability** page appears.
2. To define read capability for the endpoint metadata object, select **Enable Read Capability**.
3. Select whether the connector supports filter operations when the connector reads from the data source. Select **Platform** to specify operators and expression syntax recognized by the data loader task for the filter operation.

   Note: When a connector supports the platform filter, you can use the default operators such as `=`, `!=`, `>`, `>=`, `<`, and `<=` as filter conditions.
4. Select whether the connector supports sort to retrieve data from the data source in a specific order.
5. To add read capability attributes, click **Add** and add the attribute properties.
6. Click **Generate Code**.
   After you define the connector metadata, the Informatica Connector Toolkit generates the `<NMOName>MetadataAdapter.java` file in the **Metadata** folder.
7. Implement the following methods in the `<NMOName>MetadataAdapter.java` file to import metadata:
   populateObjectCatalog()

   Populates metadata details in the import wizard for the connector consumer.

   populateObjectDetails()

   Gets metadata from the data source based on the import dialog options settings.

Note: If you regenerate code for the endpoint metadata definition project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica perspective. You have to manually edit the source code and make changes if you add, remove, or change the endpoint metadata attributes.

Edit Endpoint Metadata Object

You can edit the endpoint metadata to update the metadata properties and add additional metadata information for records.

1. Click **Edit** for the endpoint.
2. Enter the endpoint metadata details.
   The following table describes the properties:

Property	Description
Name	Name for the endpoint metadata.
Display Name	Display name for the endpoint metadata.
Description	Description of the endpoint metadata.

3. Click **Next**.
4. To add additional metadata information for records, select **Add Record Extension** and add the following properties for the attribute:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the connection attribute.
Description	Description of the connection attribute.
Data Type	Data type of the connection attribute.
Default Value	Default value for the attribute.
Min Length	Minimum length for the value of the attribute.
Max Length	Maximum length for the value of the attribute. The maximum length of any attribute cannot be greater than Integer.MAX_VALUE, which is 2,147,483,647.
Encrypted	Indicates whether you can encrypt the attribute.
Mandatory	Indicates whether a connection attribute requires a value. If you set the Mandatory property to True but you do not want to display the attribute on the connection management user interface, you must set a default value for the attribute.
Hidden	Determines whether you want to display or hide the attribute.
Allowed Values	List of values allowed for the attribute.

5. Click **Next**.
   The **Endpoint Metadata Field** page appears.
6. To add additional metadata information for fields, select **Add Field Extension** and add the attribute properties.

7. Click **Save**.
8. Click **Generate Code**.

Package the resource or configuration files

You can package the resource or configuration files as part of the project jar.

Create a resources folder inside the project, and place the resource files.

For example, If you want to add a configuration file `custom_config.xml` for the metadata.adapter project, create a resources folder and place the configuration file in the following location:

```
<eclipse workspace>/<project_name>/usr/metadata.adapter/src
```

When you build, test, and export the connector, the configuration file is included as part of the metadata adapter jar.

Enable metadata logger

You can enable the logger to log messages in the metadata phase.

The following sample code shows the message string passed to the info() method of the Logger class:

```
logger.info("Metadata Phase in progress");
```

The messages are logged in the tomcat logs in the following directory of the agent machine:

```
<ICT installation directory>/apps/Data_Integration_Server/logs/tomcat
```

Test Metadata from the Data Source

After you define the endpoint metadata objects, you can test metadata that you import from the data source.

To debug when you test metadata from the data source, use the same debug configuration that you used to test the connection to the data source. You can also set breakpoints in the code that you want to debug.

After you define the debug configuration, you can launch the **Test Metadata** dialog box to test the connection definition and test the metadata import from the data source.

1. In the **Connector Progress** view, select Test<NMOName> under the **Test Metadata** section.

   The **Test Metadata** dialog box appears with the connection attributes that you defined for the connector.
2. Enter values for the connection attributes to connect to the data source.
3. Click **Connect**.

   The connector retrieves the metadata from the data source and the metadata appears in the **Test Metadata** page.
4. Browse and verify the retrieved metadata.
5. Click **Close**.

Step 5. Implement the Connector Run-time Behavior

Use the Informatica Connector Toolkit to implement the connector run-time behavior in Java. The connector run-time behavior defines how the connector performs operations, such as to establish a connection, close a

connection, prepare SQL statements, and run SQL statements. You can use the **Run-Time Implementation** wizard in the Informatica Connector Toolkit to implement the run-time behavior in Java.

1. In the **Connector Navigator** view, right-click the project and select **Runtime** > **<NMOName>** > **Set Up**. The **Run-Time Implementation** wizard appears.
2. Add the required run-time library files and generate the `<ConnectorID><NMOName>DataAdapter.java` and `<ConnectorID>DataConnection.java` files.
3. Implement the following methods in the `<ConnectorID><NMOName>DataAdapter.java` file:
   - initDataSourceOperation. Implement this method to perform tasks before the mapping runs. For example, you can implement code to initialize environment variables. The scope of the RuntimeConfig and Metadata handles available in this method is within the initDataSourceOperation method.
   - deinitDataSourceOperation. Implement this method to perform tasks after the mapping runs. The scope of the RuntimeConfig and Metadata handles available in this method is within the deinitDataSourceOperation method.
   - initDataSession
   - deinitDataSession
   - read
   - write
   - reset. Optional. Implement this method if the connector supports the lookup operation.
   - beginDataSession
   - endDataSession
4. Implement the following methods in the `<ConnectorID>DataConnection.java` file:
   - connect
   - disconnect

Create Messages

You can use the Informatica Connector Toolkit to create, edit, or delete messages and handle exceptions that occur during the design time or run time of the connector.

When you create messages, you specify the message text and message code, and you can include information on the message severity, cause, and user action. After you create messages, you can implement the code to handle exceptions. When you implement the code to handle the exception, you pass the message as an argument to the exception handing method.

Create design-time messages to handle design-time exceptions, such as service exceptions. Create run-time messages to handle run-time exceptions.

To create messages, perform the following steps:

1. In the **Connector Navigator** view, right-click the project and select **View and Create Messages**. The **Messages** dialog box appears.
2. Click **Add**.
   The **Add New Message** dialog box appears.
3. Enter an ID for the message.
4. Enter a code for the message. At run time, the message code and the message text appears in the session log.
5. Specify the severity of the message.

6. Enter the message text.
   You can include parameters in the message text and specify the parameters in Java message format.

   The following example shows parameters used in Java message format:

   Connection User [{0}], Port [{1,number,integer}], Connection time [{1,number}] milliseconds
7. Enter a description for the message.
8. Enter the cause of the error message.
9. Enter the suggested user action when the user encounters the error.
10. Specify whether the message is a design-time message or a run-time message.
11. Click **OK**.
    The **Messages** dialog box appears.
12. Click **Finish**.
    The message XML files appear under the **Message** folder in the **Connector Navigator** view.

Implement Design-Time Messages

You can use the methods in the com.informatica.sdk.exceptions.ExceptionManager class to implement design-time messages.

To enable localization of messages, implement the createNlsAdapterSDKException() method in the ExceptionManager class.

The following sample code shows the message parameters passed to the createNlsAdapterSDKException() method:

```
ExceptionManager.createNlsAdapterSDKException(ExceptionManager.createNlsAdapterSDKExcepti
on(MessageBundle.getInstance(),
Messages.Test_CONN_SUCC_200, "admin", 5040, 138.76);
```

To implement design-time messages that do not require localization, implement the createNonNlsAdapterSDKException() method in the ExceptionManager class.

The following sample code shows the message string passed to the createNonNlsAdapterSDKException() method:

```
ExceptionManager.createNonNlsAdapterSDKException("Unknown error:" + e.getMessage());
```

Implement Run-Time Messages

To implement run-time messages that require localization, implement the logMessage() helper method in the <ConnectorID><NMOName>DataAdapter.java file. The logMessage() method logs messages to the session log.

The following Java sample code shows the message parameters passed to the logMessage() method:

```
logMessage(Messages.Test_CONN_SUCC_200, "admin", 5040, 138.76);
```

The following C sample code shows the message parameters passed to the INFAADPLogMessage() method:

```
INFAADPLogMessage(infaDataSessionHandle, INFA_MSG_ERROR,INFA_TRACE_NONE,
CONN_ID,"admin",12,12.3);
```

To implement the run-time messages that do not require localization, implement the logger.logMessage() method.

Step 6. Test the Read Capability of the Connector

When you test the read capability of the connector, you test the connection definition, metadata of the data source, and operations that the connector supports. After you specify the test settings and run the test, you can view the result of the read operation, read operation statistics, and the log file. You can test the read capability of the connector only for the Windows platform.

To debug the code, use the same debug configuration that you used to test the connection and metadata components of the connector. You can also set breakpoints in the code that you want to debug.

After you define the debug configuration, you can launch the **Test Read** dialog box to test the read capabilities of the connector.

1. In the **Connector Progress** view, select the endpoint metadata object that appears under **Test Read**.

   The **Test Read** dialog box appears with the default JVM environment settings and tracing level. The Informatica Connector Toolkit uses the JVM settings to run the debug configuration.
2. If required, edit the JVM environment settings. Ensure that you use the same port number that appears in JVM settings for the connection properties in the debug configuration.
3. Select the required tracing level. The default is normal. Based on the amount of detail that you require in the log file, you can override the default tracing level.

   You can set the following types of tracing level:

   None

   Does not override the default tracing level.

   Terse

   Logs initialization information and error messages and notification of rejected data.

   Normal

   Logs initialization and status information, errors encountered, and skipped rows due to transformation row errors. Summarizes mapping results, but not at the level of individual rows. Default is normal.

   Verbose Initialization

   In addition to normal tracing, logs additional initialization details, names of index and data files used, and detailed statistics.

   Verbose Data

   In addition to verbose initialization tracing, logs each row. You can also get detailed statistics on where string data was truncated to fit the precision of a column.
4. Click **Next**.

   The connection attributes that you defined for the connector appears.
5. Enter values for the connection attributes to test the connection to the data source.
6. Click **Connect**.

   The **Test Metadata** page appears with the metadata imported from the data source.
7. Select the endpoint metadata objects and the corresponding endpoint metadata fields to test the read operation.
8. Click **Next**.

   The **Filter Condition** page appears.
9. Select the **Configure Filter** option in the **Filter Condition** page and specify the filter condition.

- To specify an Informatica platform expression for the filter operation, perform the following steps:
    1. In the **Definition** section, click **Add** to add an Informatica platform expression.
    2. In the **Field** column, select the field to use in the expression.
    3. In the **Operation** column, select a conditional operator to use in the expression.
    4. In the **Value** column, enter a value for the conditional expression.
- To specify a native expression for the filter operation, enter the expression in the **Definition** section.

10. After you specify expressions for the endpoint metadata object, click **Next**.

    The **Read Capability** page appears.
11. Specify values for the read capability attributes, and then click **Run**.

    The **Result** page appears. You can view the result of the read operation, read operation statistics, and the log file in the **Result** page.
12. Click **Close**.

Step 7. Publish the Connector

Use the Informatica Connector Toolkit to publish the connector. You can export the connector files to a location or publish the connector on a data loader task.

1. In the **Connector Navigator** view, right-click the project, and select **Publish**.
   The **Publish Informatica Connector** dialog box appears.
2. Click **Browse** and specify a location to publish the connector.
   The **Publish Location** defaults to the connector project workspace.
3. To publish the connector for a data loader task, specify a the plugin ID and connector version.
   If the connector package already exists, the following warning is displayed:

   ```
   Connector package with same version exists in the publish location. Click on override connector package to proceed or change the connector version.
   ```
4. Select **Override Connector Package** to override the existing package or change the connector version.
5. Click **Next** to view the contents of the client and server bundles.
6. Click **Finish**.

When you publish the connector, the Informatica Connector Toolkit bundles the connector artifacts and generates the connector package that you can use for Cloud Data Loader.

If you encounter any error or fail to publish the connector, refer to the `<connector_name>_codebuilder.log` file for more information. The `<connector_name>_codebuilder.log` file is available in the `<Eclipse workspace>/<connector project>` folder.

When you publish the connector, the Informatica Connector Toolkit restarts the Secure Agent. The Secure Agent can connect to the data source and you can create connections to the data source in a data loader task. If you fail to publish the connector, install the connector manually.

Naming Convention

A naming convention makes it easy to identify the files that belong to a connector.

Use the following guidelines when you name the connector files:

- Determine a name for the connector. Use the connector name as a prefix for the connector file names. The connector name is an alphanumeric string that can include the uppercase and lowercase letters A to Z and the numbers 0 to 9. The first character of the name must be a letter.

- Determine a unique name to identify the company building the connector. The company name is included in the package name for the connector classes.
- Use the connector name when you create a directory for the connector in the Informatica directory:

  `INFA_HOME\plugins\dynamic\`*`ConnectorName`*

The following table lists the recommended naming convention for connector files:

Component	Naming Format
Package for the connector definition and type system classes	com.vendorname.connectorname.connection
Connector definition class	ConnectorNameDefn
Type system class	ConnectorNameTypeSystem
Resource file for the connection management user interface	ConnectorNameBundle.properties or ConnectorNameBundle_lang.properties

Step 8. Test the Connector

After you create and export a new connector, you can test the connector using the automation framework. The framework generates and runs the test cases.

After you export a connector, perform the following tests:

- Code Acceptance Test (CAT) or Unit Test.
  Units tests are standalone tests. You can generate the test case after you upload data in the project explorer. After the test run, a status report is generated without any external dependency.
- Product Acceptance Test (PAT) or Integration Test.
  Integration tests require the configuration details to generate and execute the test cases. It requires user intervention for manual upload of data to the pod.

When the automation framework runs, it performs the following tasks:

1. Analyzes the connector package and generates the test suite. You can use the test suite to test the functionalities in a connector. For example, you can test the connection, verify the record and field details, search records, and test the read, write, filter operations.
2. Runs the test cases to validate the unit and integration tests. You can run the test cases separately.
3. Populates a single unified test report for both the unit and integration tests. You can see the PASS and FAIL status in the report.

CHAPTER 4

Connection Attributes

This chapter includes the following topics:

Connection Attribute Overview

The connector connection attributes determine the behavior and capabilities of a connector when it connects to a data source.

The Relational Data connector API provides a set of pre-defined connection attributes that you can use for the connector. You can enable any of the pre-defined attributes to use for the connector. When you enable a pre-defined attribute, you can set its default value or mark it as a required attribute. You can disable attributes that are not applicable to the relational database you want to connect to.

You can also define custom connection attributes for the connector. Define a custom attribute if the relational database has a connection requirement that is not represented by one of the pre-defined attributes.

Connection Attribute Properties

For each connection attribute that you define for a connector, you can set a number of properties. The properties allow you to specify whether an attribute is required, provide the default, maximum, and minimum values, or specify a list of possible values for the attribute. You can override these properties for pre-defined attributes or set these properties for custom attributes.

All connection management user interfaces, including command line programs, validate the values set for the properties of a connection attribute.

The following table describes the properties of a connection attribute:

Attribute Property	Description
isUsed	Indicates whether the connector uses the connection attribute. Set to true to include the connection attribute in the connector. Set to false to exclude the connection attribute from the connector. If this property is set to false, the connection attribute will be ignored by the connector. The attribute values cannot be set or validated and the attribute cannot be displayed on the connection management user interface.
attributeDescription	Description of the connection attribute.
isMandatory	Indicates whether a connection attribute is required and must have a value. Set to true if the connection attribute is required. Set to false if the attribute value can be null. If you set the isMandatory property to true but you do not display the attribute on the connection management user interface, you must set a default value for the attribute.
defaultValue	Default value for the attribute.
maxLength	Maximum length of a character string.
minLength	Minimum length of a character string.
minRangeValue	Lower limit of a numeric range.
maxRangeValue	Upper limit of a numeric range.
validList	List of values allowed for the attribute.
attributeCLIDisplayName	Option name for the connection attribute. This name is used when the attribute is passed as an option to an Informatica command line program. The display name cannot contain spaces.

CHAPTER 5

Type System

This chapter includes the following topics:

Type System Overview

A type system is a framework that specifies characteristics of data types. The connector type system must define the data types that the data source supports, the semantic category that matches the data type, and how they map to the data types that the Informatica platform supports.

The type system you define for a connector consists of the following sets of data types:

- Native types. Data types that the data source supports.
- Semantic types. Semantic category that matches the native data type. For example, the semantic type Length matches the native data types, such as Char, Varchar2, Binary, Varbinary, Blob, and Clob.
- Informatica platform types. Data types that the Informatica platform supports. The Informatica Connector Toolkit API uses ODBC as a model for describing Informatica platform data types.

Native Types and Semantic Categories

The native types included in the type system are all possible data types in the data source for which the connector is built.

You must define the semantic category of each data type in the data source. The Informatica Connector Toolkit API defines each semantic category. Check the data source documentation to verify the data types that are available in the data source.

Use the Informatica Connector Toolkit to associate native data types with semantic categories. The data type name must match the character string that corresponds to the type name returned during the metadata import process, such as Integer, Varchar2, or Blob.

When you define the semantic category for a data type, you can modify the precision and scale returned by the import process so that the data type matches the requirements of the type system.

Use the following semantic categories to classify the native types:

Length semantics

Use this category for native types where length is the principal characteristic. This category can include data types such as Char, Varchar2, Binary, Varbinary, Blob, and Clob.

Integer semantics

Use this category for native types that can contain signed integers. The length of the data type is the number of decimal digits specified in the data type precision. This category can include data types such as Integer, Smallint, Bigint, and Tinyint.

Machine integer semantics

Use this category for native types that can contain signed or unsigned integers. The length of this data type is measured in bytes. The precision of a machine integer type is the maximum number of decimal digits that fits within the length of the data type, regardless of whether all possible values can be stored. For example, a 32-bit (4 byte) machine integer can store up to 10 digits but if the value of each digit is 9, then the value of the integer can result in an overflow.

Decimal semantics

Use this category for native types that can contain an exact real number where precision is the total number of digits and scale is the number of digits to the right of the decimal point. The precision for this semantic category must be greater than or equal to the scale. This category can include data types such as Decimal and Numeric.

Scientific decimal semantics

Use this category for native types that can contain an exact real number where precision is the number of digits stored rather than the total number of digits represented by the number. The total number of digits represented by the number can exceed the precision. The scale of the data type can exceed precision and can be positive or negative. A positive scale represents digits to the right of the decimal point. A negative scale represents the rounding position to the left of the decimal point. This category can include data types such as the Number data type in Oracle with precision and scale specified.

Float semantics

Use this category for binary or decimal floating point data types. An example of a binary floating point type is the binary_float type in Oracle. An example of a decimal floating point number is the Number data type in Oracle with precision and scale not specified.

Gregorian date semantics

Use this category for date types that the connector can expose as Gregorian dates, times, and timestamps.

Native Type Properties

Depending on the semantic category, you must set one or more of the following native type properties:

- Maximum precision
- Minimum precision
- Default precision
- Support for changes to precision
- Maximum scale
- Minimum scale
- Default scale
- Support for changes to scale

- Unit of length, such as characters, bytes, and bits
- For Gregorian date semantics, support for hour, minute, second, year, month, day, or time zone
- For float semantics, radix of the precision and exponent

Set the maximum and minimum values for precision and scale to validate the following:

- Imported database metadata
- Data type of a column that an end user adds to a table in a mapping

The default precision, default scale, and the specification of whether the precision and scale can be modified apply to metadata that is manually defined by the user. If the precision or scale cannot be modified, then the default value and the maximum value must be set to the same value.

Cloud Data Integration Types

Cloud Data Integration types include all the data types that are recognized by Informatica. The Informatica Connector Toolkit API uses ODBC as a model to describe the Cloud Data Integration data types.

In addition to assigning semantic categories to native types, you must match each semantic category with one or more Cloud Data Integration data types. The Cloud Data Integration uses ODBC types to determine what type of data buffers to bind to the run-time connector for data access. In some cases, the match between a semantic type and a Cloud Data Integration type is not exact. For example, the Cloud Data Integration always binds a timestamp buffer if the semantic category is mapped to an Cloud Data Integration date, time, or timestamp data type.

In many cases, a semantic category matches one of the available Cloud Data Integration types. In cases where there is no exact match between types, select the best possible match.

In addition to defining the best mappings to convert native types to Cloud Data Integration types, you can specify alternate mappings. An alternate mapping allows the end user to select alternate transformation data types to associate with the input and output ports of a data source used in a mapping.

Use the Informatica Connector Toolkit to add matching Cloud Data Integration data types for the semantic category. You can indicate whether the mapping is the exact match or whether the mapping can result in a lossy data conversion due to a match that is not exact.

Note: Cloud Data Integration types are not the same as the transformation data types that appear in a transformation that you add to a mapping. Cloud Data Integration types are internal data types used only within the connector type system. Informatica can convert Cloud Data Integration types to transformation data types.

CHAPTER 6

Metadata Objects

This chapter includes the following topics:

Connector Metadata Overview

Define the Connector metadata components to represent the metadata of the data source. After you define the metadata components, you can fetch and display the metadata in Cloud Data Integration with information on the native data type, precision, and scale.

Use the Informatica Connector Toolkit to define one or more native metadata definitions for the Connector to read from and write to the data source. The Informatica Connector Toolkit internally uses a metadata model that represents the metadata.

Metadata Components

The metadata model contains components that represent the metadata of the data source.

The metadata model consists of the following components:

Flat Record

A flat record represents a structure that contains columns, unique keys, and primary keys. The structure of a flat record is similar to a database table that contains columns and keys.

A flat record contains attributes that store the following information:

- Name of the endpoint metadata object
- Type of endpoint metadata object
- Related records for the flat record
- Primary key for the flat record
- Unique keys for the flat record
- Indexes for the flat record

- Any additional record attributes specific to the data source

Field

A field is a data structure for a single unit of data in a data source.

A field contains attributes that store the following information:

- Name of the field
- Default value of the field
- Precision of the field
- Scale of the field
- Boolean value that indicates whether the field can contain a null value
- Any additional field attributes specific to the data source

Constraints

Constraints represent the primary key and unique keys for a flat record.

The primary key and unique key contain attributes that store the following information:

- Name of the key
- Native name of the key defined in the native metadata
- List of fields that form the key

Index

Index represents a native index that orders the flat records or uniquely identifies a row in the flat record.

An index contains attributes that store the following information:

- Name of the index
- Native name of the index
- Boolean value that indicates whether the index is unique
- List of index fields
- Index order to retrieve the data

Import Dialog Box Settings

Use the Informatica Connector Toolkit to define the import options that appear in Cloud Data Integration when a connector consumer creates a data object.

The following tables describes the import options:

Option	Description
Allow Multi Select	Allows selection of multiple importable objects. Default is false.
Display Filter By Name	Displays the filter by name option. Default is true.
Display Filter By Description	Displays the filter by description option. Default is true.

Option	Description
Display Filter By Path	Displays the filter by path option. Default is true.
Display Skip Description	Displays the skip descriptions check box. Default is true.
Show Entity	Shows entity details. Default is true.
Show Hierarchy	Shows metadata hierarchy. Default is true.
Show Related Records	Shows related records. Default is true.

CHAPTER 7

Partitioning Capability

This chapter includes the following topics:

Partition Capability Overview

You can use the Informatica Connector Toolkit to specify the partition type and implement the partition logic to use when the Cloud Data Integration reads or writes data.

Based on the partition logic you implement for a connector, the Data Integration Service dynamically divides the underlying data into partitions and processes all of the partitions concurrently.

You can specify the following partitioning types for a connector:

Dynamic

A partitioning logic that determines the number of partitions at run time based on the partition information from the data source. If the data source provides partition information, you can use dynamic partitioning to increase the performance of connector read and write operations.

Static

A partitioning logic that is based on the partition information that the user specifies, such as number of partitions or key range. If you require the user to specify the partition information, you can implement fixed partitioning for the connector. If the tables in the data source support key range partitioning, you can add key range partitioning capability for the connector.

Automatic Partitioning

You can use the Informatica Connector Toolkit to configure a connector to dynamically determine partition information from the data source. Connectors with dynamic partitioning capability do not require partition information from the user when the connector reads or writes data.

If you implement dynamic partitioning for a connector, the connector queries the data source for the number of source or target partitions and other partition-specific attributes. For example, if the data source is a

relational database, you can make use of the partition information from the database to implement the partition logic.

When you define a endpoint metadata object, you can implement dynamic partitioning for both read and write capabilities of the endpoint metadata object.

Static Partitioning

You can use the Informatica Connector Toolkit to configure static partitioning if you require the user to specify the partition information before the connector reads data.

When you define a endpoint metadata object, you can implement static partitioning for read capabilities of the endpoint metadata object. Based on the data source, you can implement following static partitioning types:

Fixed

If you require partition logic based on the partition information specified by the user, implement fixed partitioning capability. For example, if the data source does not provide partition information, you can implement partitioning logic based on the user inputs. The user enters the partition information, such as the number of partitions, before the connector reads data from the data source.

Key range

If the tables in the data source support key range partitioning, you can add support for key range partitioning capability. Before you add support for key range partitioning, you must ensure that the connector supports filter operation and platform expression. The Informatica Connector Toolkit implements key range partitioning as a filter query. The connector user enters the partition keys and key range when the connector reads data from the data source.

You can implement static partitioning only for connectors with read capability.

CHAPTER 8

Pushdown Capability

This chapter includes the following topics:

Pushdown Capability Overview

You can use the Informatica Connector Toolkit to configure pushdown logic for a connector to push transformation logic to source or target databases for processing using the native connection.

You can specify the following pushdown types for a connector:

Full

A pushdown logic where the task pushes as much of the transformation logic as possible to the target database.

Source

A pushdown logic where the task pushes down as much as the transformation logic as possible to process in the source database. When you select source pushdown optimization, the task pushes the transformation logic for all the supported transformations downstream in the mapping.

Note: The Informatica Connector Toolkit currently does not implement the source pushdown logic for a connector.

Pushdown Optimization Execution Flow

You can use the Informatica Connector Toolkit to configure full pushdown using the native database connection to push the entire mapping logic to target databases for execution.

The Informatica Connector Toolkit performs pushdown optimization with the following execution flow:

1. During runtime, the source and the target connectors are invoked to validate if the current transformation is supported. You must declare the support for pushdown to the source in `adapter.contribution.plugin.xml` file.

2. When source pushdown support is enabled, the transformation might get pushed to the source database instead of the target database. An additional flag is marked in the intermediate result to indicate that source handles the transformations.
3. Additional validation callback configured in the target connector checks if the target database can support the current transformation. If the target connector returns true, the flag in the intermediate result is updated to indicate that target performs full pushdown provided the forward data flow reaches the target.
4. If the data flow does not reach the target and the additional flag is marked in the intermediate result in Step 2, the source database will handle the transformations.
   You cannot push partial mapping logic to the target database using source pushdown.
5. If the forward data flow reaches the target, the mapping is updated to replace the source with a placeholder source and the target transformation contains a new data object with a reference to the source object. If the data flow does not reach target, the mapping runs without pushdown. You cannot partially push down transformation logic with full pushdown.

Classes and Methods for Pushdown Capabilities

After you define the connector metadata for pushdown capability and generate the code, the Informatica Connector Toolkit adds the pushdown specific parameters to the `adapter.contribution.plugin.xml`.

Changes to adapter.contribution.plugin.xml

Informatica Connector Toolkit adds the following section to the adapter.contribution.plugin.xml file:

- The SourceToTargetPushdown section defines the connections for which you want to support full pushdown.
  The SourceToTargetPushdown section is populated with the following attributes as defined in the connector metadata:
  - adapterID. Specifies the connector ID of the source.
  - sourcePushdown. Set to true if you enable **Support Source Pushdown** in the connector metadata.

    Note: The Informatica Connector Toolkit currently does not implement the source pushdown logic for a connector.
  - multiTargetSupport. Set to true if you enable **Support Multi Target** in the connector metadata.
- The InputProjectionSupport section defines the support for full pushdown for different transformations.
  The InputProjectionSupport section includes the following elements:
  - FilterOperation
  - JoinOperation
  - ExpressionOperation
  - AggregatorOperation
  - LookupOperation
  - SortOperation

  If you enable full pushdown for a transformation in the connector metadata, the Informatica Connector Toolkit sets the supportsPushdown attribute to true for the respective transformation element.

runtime.pdo

The Informatica Connector Toolkit provides the following classes and interfaces to implement pushdown logic:

- Renderer Class
  This class converts the ASG to an SQL query string by traversing each node type. Each node type represents a query concept. Each rendered node type will return a string. Once all the node types are traversed successfully, a pushdown SQL Query is generated.
- Visitor Class
  The connector can override the visitXXXNode functions to generate a connector specific query. The expression visitor functions are not directly invoked. These functions are invoked when connector runs the utility API such as `SDKPushdownUtils::getPushdownSQL` either at runtime to generate the connector specific SQL query or during the validation phase to check if the connector supports the generated query.
- TypeHandler Class
  Implements the typeHandler interface to create the connector context and type conversion handler that is used to generate the pushdown query.
- TypeConversionHandler Class
  This class implements the TypeConversionHandler interface to handle the conversion from the platform type to the AdapterType or from the Adapter native type to the AdapterType.
- TypeContext Class
  The TypeContext captures the container typesystem context belongs to the platform type or the AdapterType.
- Pair Class
  The Pair class and the respective functions are used in the ASG traversal. Do not modify this class or any function in this class.
- AdapterType Interface
  Native typesystem is a limited or static interface based on proprietary Informatica APIs and XML based registries. It does not allow you to control the type semantics programmatically using APIs such as `AdapterType::isA` or add plain old java objects such as `public static class Varchar implements AdapterType`.

  AdapterType is a part of SDkQueryNode hierarchy.
- Engine Class
  The Engine class implements SDKEngine interface that helps you to identify the engine type for mappings and elastic mappings
- Post Proc Class
  The Post Proc class extends the Renderer class to traverse ASG and generate the SQL query string. It also optimizes the pushdown query.

For more information about implementing these classes, see the sample source code of the MySQL_Cloud Connector available with the Informatica Connector Toolkit.

CHAPTER 9

Mappings in advanced mode

This chapter includes the following topics:

Mappings in advanced mode overview

You can create a mapping in advanced mode. In advanced mode, the Mapping Designer updates the mapping canvas to include transformations and functions that enable advanced use cases.

To run mappings in advanced mode, enable the read and write capability for advanced mode when you build a connector.

Project, classes, and methods

After you define the connector metadata for pushdown capability and generate the code, the Informatica Connector Toolkit creates the runtime.spark project with DataFrameAdapter.scala class.

When you enable the read and write capabilities for mappings that run on the advanced cluster, you must implement the read and write methods in the Informatica Connector Toolkit scala class DataFrameAdapter.scala.

The following code snippet shows an example of the Scala class with read and write methods for MySQL_Cloud Connector:

```
package com.infa.adapter.msql.runtime.scalatask

import java.util.Properties

import com.informatica.sdk.adapter.metadata.projection.semantic.iface.Operation
import
com.informatica.sdk.adapter.metadata.projection.sinkoperation.semantic.iface.NativeSink
import
com.informatica.sdk.adapter.metadata.projection.sinkoperation.semantic.iface.PlatformSink
import com.informatica.sdk.scalatask.dataframe.AdapterContext
import com.informatica.sdk.scalatask.dataframe.DataFrameAdapter
import
com.informatica.sdk.adapter.metadata.capabilityattribute.semantic.iface.ASOComplexType
import
com.informatica.sdk.adapter.metadata.capabilityattribute.semantic.iface.ReadCapabilityAtt
ributes
```

```
import
com.informatica.sdk.adapter.metadata.capabilityattribute.semantic.iface.WriteCapabilityAt
tributes
import
com.informatica.sdk.adapter.metadata.common.datasourceoperation.semantic.iface.Capability
import
com.informatica.sdk.adapter.metadata.common.datasourceoperation.semantic.iface.ReadCapabi
lity
import com.informatica.sdk.adapter.metadata.patternblocks.field.semantic.iface.Field
import
com.informatica.sdk.adapter.metadata.patternblocks.flatrecord.semantic.iface.FlatRecord
import
com.informatica.sdk.adapter.metadata.projection.expression.semantic.iface.BinaryOperatorE
num
import com.informatica.sdk.adapter.metadata.projection.expression.semantic.iface.Constant
import
com.informatica.sdk.adapter.metadata.projection.expression.semantic.iface.DateConstant
import
com.informatica.sdk.adapter.metadata.projection.expression.semantic.iface.DecimalConstant
import
com.informatica.sdk.adapter.metadata.projection.expression.semantic.iface.FieldIdentifier
import
com.informatica.sdk.adapter.metadata.projection.expression.semantic.iface.IntegerConstant
import
com.informatica.sdk.adapter.metadata.projection.expression.semantic.iface.StringConstant
import
com.informatica.sdk.adapter.metadata.projection.expression.semantic.iface.SDKExpression
import
com.informatica.sdk.adapter.metadata.projection.filteroperation.semantic.iface.FilterOper
ation
import
com.informatica.sdk.adapter.metadata.projection.joinoperation.semantic.iface.JoinOperatio
n
import
com.informatica.sdk.adapter.metadata.projection.projectionoperation.semantic.iface.Projec
tionOperation
import
com.informatica.sdk.adapter.metadata.projection.projectionoperation.semantic.iface.Select
TypeEnum
import com.informatica.sdk.adapter.metadata.projection.semantic.iface.FieldBase
import com.informatica.sdk.adapter.metadata.projection.semantic.iface.NativeField
import com.informatica.sdk.adapter.metadata.projection.semantic.iface.OperationBase
import
com.informatica.sdk.adapter.metadata.projection.simpleexpression.semantic.iface.SimpleBin
aryExpression
import
com.informatica.sdk.adapter.metadata.projection.simpleexpression.semantic.iface.SimpleSDK
Expression
import
com.informatica.sdk.adapter.metadata.projection.sortoperation.semantic.iface.SortOperatio
n
import
com.informatica.sdk.adapter.metadata.projection.sortoperation.semantic.iface.SortOrderEnu
m
import
com.informatica.sdk.adapter.metadata.projection.sourceoperation.semantic.iface.NativeSour
ce
import com.informatica.sdk.scalatask.dataframe._

import java.text.MessageFormat
import java.math.BigDecimal
import java.math.BigInteger
import java.text.SimpleDateFormat
import java.util.ArrayList
import java.util.logging.Logger
import java.util.logging.Level
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.SparkSession
import org.apache.spark.SparkContext
import org.apache.spark.sql.functions._
```

```
import org.apache.spark.sql.types.DateType
import com.informatica.sdk.adapter.metadata.aso.semantic.iface.ASOOperation

import scala.collection.JavaConverters._
import com.informatica.sdk.adapter.metadata.extension.semantic.iface.KeyValueExtension
import com.informatica.sdk.adapter.metadata.projection.semantic.iface.Projection
import org.apache.spark.sql.streaming.StreamingQuery

import scala.collection.mutable.ListBuffer;
// * This represents the spark pushdown specific class that can be extended by adapters
to provide optimized spark pushdown for
// * providing/consuming dataframes
// */

class MSQLDataFrameAdapter extends DataFrameAdapter {

  private var JDBC_DRIVER ="com.mysql.jdbc.Driver";
  private var asoOp: ASOOperation = null;
  private var logger: Logger = null;
  private var tableName: String = "";
  private var connectedFields: ArrayList[FieldInfo] = null;
  private var isTruncateTarget: Boolean = false;
  private val TRUNCATE_TARGET: String = "truncateTargetTable";

  private var nativeRecords = new ListBuffer[FlatRecord]();

   /*
   This method reads data from the external data source and
   returns a dataframe containing source data
   Sample read scala code for mysql is provided in this method
   */
   override def read(sparkContext:SparkContext, adpContext: AdapterContext) : DataFrame={
            //Initialize the mysql jdbc driver
            Class.forName("com.mysql.jdbc.Driver");
            //Fetch the logger from the AdapterContext
            var logger = adpContext.getLogger();
            //Get the asooperation
            var asoOp = adpContext.getASOOperation();
            //Fetch te connection attributes
            var connAttrs = adpContext.getConnectionAttributes();
            var query: String = null;

            //Initialize all the connection attributes to variables
            var user:String = connAttrs.get("username").asInstanceOf[String];
            var password:String = connAttrs.get("password").asInstanceOf[String];
            var host:String = connAttrs.get("host").asInstanceOf[String];
            var port:Int = connAttrs.get("port").asInstanceOf[Int];
            var catalog:String = connAttrs.get("catalog").asInstanceOf[String];
            var connectionURL:String=connAttrs.get("connectionURL").asInstanceOf[String];

            logger.log(Level.INFO, "Read scala method invoked");

            //Generate the mysql jdbc url using connection attributes
            if (catalog !=null || !catalog.isEmpty()) {
                connectionURL= "jdbc:mysql://" + host + ":" + port + "/" + catalog;
            }else {
                connectionURL= "jdbc:mysql://" + host + ":" + port;
                val catalog="";
            }
            //Create properties object with all the connection attributes
            val connProps = new Properties
                    connProps.put("user",s"${user}");
            connProps.put("password",s"${password}");
            connProps.put("Driver",s"${JDBC_DRIVER}");

          //Fetch the projection list from the asooperation
           val projList:List[Projection] = asoOp.getASOProjectionsList().asScala.toList;
           val p: Projection = null
           val ob: OperationBase = null
           var fieldsList: List[FieldBase] = null
```

```
      //Creates a list of source fields from the projection
      for (p <- projList) {
        for (ob <- p.getBaseOperations.asScala.toList) {
          if (ob.isInstanceOf[PlatformSink]) {
            val inp: OperationBase = ob.asInstanceOf[PlatformSink].getInputBaseOperation
            fieldsList = inp.asInstanceOf[Operation].getOperationFields.asScala.toList
          }
        }
      }

      //Creates a list of native records(sources) from the projection
      for (p <- projList) {
        for (ob <- p.getBaseOperations.asScala.toList) {
          if (ob.isInstanceOf[NativeSource]) {
            if(ob.asInstanceOf[NativeSource].getNativeRecord.isInstanceOf[FlatRecord])
            nativeRecords +=
(ob.asInstanceOf[NativeSource].getNativeRecord).asInstanceOf[FlatRecord]
          }
        }
      }

    //Use only the first native record assuming single source
    var record:FlatRecord = nativeRecords(0);

    //Fetch the source table name
    tableName=record.getNativeName();

    //Get the connected fields for the source
    connectedFields = getConnectedFields(fieldsList);

     //Generate read query using the connected fields and adapter context
        query = getQuery(connectedFields, adpContext);

      //Use the logger for logging messages to the session log
      logger.log(Level.INFO, "Reader started");
      logger.log(Level.INFO, "The run-time engine uses the following SQL query to read
data: " + query);

     //Create a spark sql context to read the data from the source
      val sqlContext = new org.apache.spark.sql.SQLContext(sparkContext)

     //Use sqlContext.read to fetch the data from the jdbc source
      val df_read: DataFrame = sqlContext.read.format("jdbc").option("url", s"$
{connectionURL}").option("driver", s"${JDBC_DRIVER}").option("query", s"$
{query}").option("user", s"${user}").option("password",s"${password}").load()

      logger.log(Level.INFO, "Reader completed");

      return  df_read;
    }

  /*
   This method writes data from the target using the data frame provided in the write
call
   Sample write scala code for mysql is provided in this method
  */
    override def write(sparkContext:SparkContext, adpContext: AdapterContext, dataFrame:
DataFrame)= {
           //Initialize the mysql jdbc driver
            Class.forName("com.mysql.jdbc.Driver");
            //Get the asooperation
            var asoOp = adpContext.getASOOperation();
            //Fetch te connection attributes
            var connAttrs = adpContext.getConnectionAttributes();
            //Fetch the logger from the AdapterContext
            var logger = adpContext.getLogger();

            //Initialize all the connection attributes to variables
            var user:String = connAttrs.get("username").asInstanceOf[String];
            var password:String = connAttrs.get("password").asInstanceOf[String];
```

```
            var host:String = connAttrs.get("host").asInstanceOf[String];
            var port:Int = connAttrs.get("port").asInstanceOf[Int];
            var catalog:String = connAttrs.get("catalog").asInstanceOf[String];
            var connectionURL:String=connAttrs.get("connectionURL").asInstanceOf[String];

            //Use the logger for logging messages to the session log
            logger.log(Level.INFO, "Write scala method invoked");

            //Generate the mysql jdbc url using connection attributes
            if (catalog !=null || !catalog.isEmpty()) {
                connectionURL= "jdbc:mysql://" + host + ":" + port + "/" + catalog;
            }else {
                connectionURL= "jdbc:mysql://" + host + ":" + port;
                val catalog="";
            }

          //Create properties object with all the connection attributes
            val connProps = new Properties
           connProps.put("user",s"${user}");
            connProps.put("password",s"${password}");
            connProps.put("Driver",s"${JDBC_DRIVER}");

      var record:FlatRecord = null
            val p: Projection = null
            val ob: OperationBase = null

            //Fetch the projection list from the asooperation
            val projList:List[Projection] = asoOp.getASOProjectionsList().asScala.toList;

            //Fetch the target record from the projection
            for (p <- projList) {
            for (ob <- p.getBaseOperations.asScala.toList) {
                if (ob.isInstanceOf[NativeSink]) {

if(ob.asInstanceOf[NativeSink].getNativeRecord.isInstanceOf[FlatRecord])
                    record =
ob.asInstanceOf[NativeSink].getNativeRecord.asInstanceOf[FlatRecord]
                }
            }
        }

        //Fetch the target table name
        val targettablename:String =record.getNativeName();
        //Fetch the write capability attributes
        var writeCap = asoOp.getWriteCapabilityAttributes();
        //Sample code to fetch the write capability attribute extensions and their values
        val writeCapExt: KeyValueExtension =
writeCap.getExtensions().asInstanceOf[KeyValueExtension];
        var keyMap: java.util.Map[String, Object] = writeCapExt.getAttributesMap();
        //Sample code to fetch the value for TRUNCATE_TARGET write capability attribute
        isTruncateTarget= keyMap.get(TRUNCATE_TARGET).asInstanceOf[Boolean];
        logger.log(Level.INFO, "Writer Started");

        if (isTruncateTarget) {
               logger.log(Level.INFO, "The truncate table property is enabled.");
               //Use overwrite mode if truncate target is enabled
               dataFrame.write.mode("overwrite").jdbc(s"${connectionURL}", s"$
{targettablename}", connProps)
        }
        else
        {
          //Use append mode if truncate target is not enabled
          dataFrame.write.mode("append").jdbc(s"${connectionURL}", s"$
{targettablename}", connProps)
        }
          logger.log(Level.INFO, "Writer Completed");
        }

  /*
   This method gets the list of connected fields
  */
```

```
    def getConnectedFields(projectionFields: List[FieldBase]): ArrayList[FieldInfo] = {
    var i: Int = 0;
    var fields: ArrayList[FieldInfo] = new ArrayList[FieldInfo]();
    var pfield = null;
    for (pfield <- projectionFields) {
      var fieldName=pfield.getName().toString();
      var f: FieldInfo = new FieldInfo(pfield, i);
      i +=1;
    }
    return fields;
  }

   /*
     This method generates the read query based on the connected fields
   */
  def getQuery(connectedFields: ArrayList[FieldInfo], adpContext: AdapterContext):
String = {
    val query = new StringBuilder();
    query.append("SELECT ");
    var addComma: Boolean = false;
    var field = null;
    val runtimeCtx = adpContext.getRuntimeContext();
    var i: Int = 0;
    for (field <- connectedFields.asScala) {
      if (addComma) {
        query.append(", ");
      }
      if (runtimeCtx.isFieldConnected(i)) {
        query.append("\"" + field.getField().getNativeSourceField().getNativeName() +
"\"");
      } else {
        query.append("NULL as " + "\"" +
field.getField().getNativeSourceField().getNativeName() + "\"");
      }
      addComma = true;
      i += 1;
    }
    query.append(" FROM " + tableName);
    return query.toString();
  }

}
```

CHAPTER 10

Manual Changes to Informatica Connector Toolkit Source Code

This chapter includes the following topics:

Manual Changes to Informatica Connector Toolkit Source Code Overview

The Informatica Connector Toolkit does not regenerate the existing source code for the following capabilities:

- Custom Query
- Object Path Override
- Full Pushdown implementation in the ASOOperationObjMgr class in the runtime.semantic package

You have to manually edit the source code to add the native metadata attributes for these capabilities.

Code Changes for Connection Pooling

When you enable connection pooling for the connector, you must manually modify the Informatica Connector Toolkit classes and methods to implement the connection pooling logic.

1. Override the validateConnection() method in the connection class of metadata connector.

The method executes a simple query to validate whether the connection is open or closed. Use this method if testOnBorrow, testOnReturn, or testOnCreate is enabled for connection pooling.

```
@Override
    public Status validateConnection() {
        // This is just a sample logic for testing, actual logic might be different
        String validateConnectionQuery = "select 1 from dual";
        Statement stmt = null;
        StatusEnum status = StatusEnum.SUCCESS;
        StringBuilder errMsgBuilder = new StringBuilder("");
        try {
            stmt = conn.createStatement();
            stmt.executeQuery(validateConnectionQuery);
        } catch (SQLException e) {
            status = StatusEnum.FAILURE;
            errMsgBuilder.append(
                    "Following error occurred while executing query  : [" +
validateConnectionQuery + "]\n[" + e.getMessage() + "].\n");
        } finally {
            try {
                if (stmt != null) {
                    stmt.close();
                }
            } catch (SQLException sqlException) {
                status = StatusEnum.FAILURE;
                errMsgBuilder.append(
                        "Error occurred while closing JDBC statement due to : [" +
sqlException.getMessage() + "]");
            }
        }

        return new Status(status, errMsgBuilder.toString());
    }
```

2. Override the isEqual() method in the connection class of metadata connector. The method compares the connection attributes of the connection objects as shown below:

```
/**
     * This api is used if customComparison is enabled for connection pooling for
comparing two connection objects
*/
    @Override
    public boolean isEqual(Map<String, Object> connAttrs) {
        if(connAttrs != null) {
            if(this.connAttrs.containsKey("username") &&
connAttrs.containsKey("username")) {
                if(!
(this.connAttrs.get("username").equals(connAttrs.get("username"))))
                    return false;
            }

            if(this.connAttrs.containsKey("password") &&
connAttrs.containsKey("password")) {
                if(!
(this.connAttrs.get("password").equals(connAttrs.get("password"))))
                    return false;
            }

            if(this.connAttrs.containsKey("host") && connAttrs.containsKey("host")) {
                if(!(this.connAttrs.get("host").equals(connAttrs.get("host"))))
                    return false;
            }

            if(this.connAttrs.containsKey("port") && connAttrs.containsKey("port")) {
                int port1 = (int)this.connAttrs.get("port");
                int port2 = (int)connAttrs.get("port");
                if(port1 != port2)
                    return false;
            }
        }
```

```
            return true;
        }
```

3. Override the generateHashCode() method in the connection class of metadata connector.
   This pool uses this method to generate the hash code for the key in the pool if the connector supports custom comparison of the connection attributes.

```
    /**
         * This API is used to generate the hashCode for the connection object. It is
    used only when custom comparison is enabled in Design time connection pooling
         */
        @Override
        public int generateHashCode() {
            final int prime = 31;
            int result = 1;
            // use only those connection attribute to generate hashcode which you are
    using in isEqual method to compare 2 Connection objects
            ArrayList<String> supportedConnAttr = new ArrayList<String>() {{
                add("username");
                add("password");
                add("host");
                add("port");}};

            if(connAttrs != null) {
                for(Map.Entry<String,Object> mapElement : connAttrs.entrySet()) {
                    Object val = mapElement.getValue();
                    String key = mapElement.getKey();
                    if(val != null && supportedConnAttr.contains(key)) {
                        result = prime * result + val.hashCode();
                    }
                }
            }

            return result;
        }
```

Connection Pooling through API

To configure connection pooling through the API, after you make the code changes for connection pooling, override the getPoolingOptions() method in the MetadataAdapter class.

The method sets the following values of the connection pooling attributes:

```
@Override
    public void getPoolingOptions(ConnectionPoolingOptions options){
            /**
             * The default values are:
             * maxTotal = -1
             * maxTotalPerKey = -1
             * maxIdlePerKey = 20
             * MinEvictableIdleTimeMillis = 1200000
             * MaxWaitMillis = 30000
             */

         /* boolean supportPooling = Boolean.FALSE;
            boolean supportPooling =
(System.getProperty(Demo4Connection_CONNECTIONPOOLING_ENABLE) == null
                    ||
System.getProperty(Demo4Connection_CONNECTIONPOOLING_ENABLE).isEmpty()) ? Boolean.FALSE
                            :
Boolean.parseBoolean(System.getProperty(Demo4Connection_CONNECTIONPOOLING_ENABLE));
            int maxTotal =
(System.getProperty(Demo4Connection_CONNECTIONPOOLING_MAXTOTAL) == null
                    ||
System.getProperty(Demo4Connection_CONNECTIONPOOLING_MAXTOTAL).isEmpty()) ? 0
                            :
Integer.parseInt(System.getProperty(Demo4Connection_CONNECTIONPOOLING_MAXTOTAL));
            int maxIdlePerKey =
(System.getProperty(Demo4Connection_CONNECTIONPOOLING_MAXIDLE_PERKEY) == null
```

```
                    ||
System.getProperty(Demo4Connection_CONNECTIONPOOLING_MAXIDLE_PERKEY).isEmpty()) ? 0
                                :
Integer.parseInt(System.getProperty(Demo4Connection_CONNECTIONPOOLING_MAXIDLE_PERKEY));
            int maxTotalPerKey =
(System.getProperty(Demo4Connection_CONNECTIONPOOLING_MAXTOTAL_PERKEY) == null
                    ||
System.getProperty(Demo4Connection_CONNECTIONPOOLING_MAXTOTAL_PERKEY).isEmpty()) ? 0
                                :
Integer.parseInt(System.getProperty(Demo4Connection_CONNECTIONPOOLING_MAXTOTAL_PERKEY));
            long minEvictableIdleTime =
(System.getProperty(Demo4Connection_CONNECTIONPOOLING_MINEVICTABLE_IDLETIME_MILLIS) ==
null
                    ||
System.getProperty(Demo4Connection_CONNECTIONPOOLING_MINEVICTABLE_IDLETIME_MILLIS).isEmpt
y()) ? 0
                                :
Long.parseLong(System.getProperty(Demo4Connection_CONNECTIONPOOLING_MINEVICTABLE_IDLETIME
_MILLIS));

            if (supportPooling) {
                options.setSupportPooling(Boolean.TRUE);
                options.setTestOnBorrow(Boolean.TRUE);
                //If value is set to zero then we will use default values
                if (maxTotal > 0)
                    options.setMaxTotal(maxTotal);
                if (maxIdlePerKey > 0)
                    options.setMaxIdlePerKey(maxIdlePerKey);
                if (maxTotalPerKey > 0)
                    options.setMaxTotalPerKey(maxTotalPerKey);
                if (minEvictableIdleTime > 0)
                    options.setMinEvictableIdleTimeMillis(minEvictableIdleTime);
            } else {
                options.setSupportPooling(false);
            }
            */
    }
```

Code Changes for Custom Query Capability

When you enable the custom query capability for the connector, you must manually modify the Informatica Connector Toolkit classes and methods to implement the custom query logic.

Add the following code changes to implement the custom query capability for the connector:

1. In the populateObjectCatalog() method of the MetadataAdapter class, validate if a custom query is used to import the data object:

```
//Sample code to check if custom query is enabled
public String getCustomQuery(List<Option> options) {

try{
        for (Option opt : options) {
            int optionID =
opt.getDescription().getEffectiveDefinition().getOptionID();

            if (optionID == CCatalogImportOpts.QUERY)
                return opt.getValue().toString();
        }
        //returns empty string or null if custom query is not enabled
}
```

2. If custom query is enabled, then import the data object by executing the query. Call the following method from populateObjectCatalog() method:

```
//Sample code to import data object by executing the query
public boolean populateCatalogForCustomQuery(Connection connection, Catalog catalog,
String customQuery){

            Factory sdkFactory = catalog.getFactory();
            FlatRecord record = null;
            Package pack = sdkFactory.newPackage(catalog);
            pack.setName("CUSTOM_CATALOG");
            pack.setNativeName("CUSTOM_CATALOG");
            catalog.addRootPackage(pack);
            record=sdkFactory.newFlatRecord(catalog);

    try {

          // sample code to execute the custom query

            Connection nativeConn = conn.getNativeConnection();
            stmt = nativeConn.createStatement();
            rs = stmt.executeQuery(customQuery);
            rsMetaData = rs.getMetaData();
            String tableName = rsMetaData.getTableName(1);
            int numberOfColumns = rsMetaData.getColumnCount();


        // Sample code to populate the object details

            record.setName(tableName);
            record.setNativeName(tableName);

            for (int i = 1; i <= numberOfColumns; i++) {
                    int scale=rsMetaData.getScale(i);
                    int precision=rsMetaData.getPrecision(i);
                    String columnName = rsMetaData.getColumnLabel(i);
                    String colType = rsMetaData.getColumnTypeName(i);
                    Field field=null;
                    field = sdkFactry.newField(catalog);
                    field.setNativeName(columnName);
                    field.setScale(scale);
                    field.setDataType(colType);
                    field.setPrecision(precision);
                    record.addField(field);
        }

    }
     catch (SQLException e) {
                ExceptionManager.createNonNlsAdapterSDKException(
                        "An error occured while executing custom query:[" +
e.getMessage() + "]");
                return false;
        }

        return true;
}
```

3. Validate the custom query specified by the user. The validate() method is executed when the user validates the custom query in the **Add Native Metadata Definition** dialog box.

   The following code snippet shows an example of validating the custom query for MySQL_Cloud Connector:

```
@Override
    public boolean validate(Connection sdkConnection, List<Option> options) {

        String customQuery = getCustomQuery(options);
        java.sql.Connection mySQLConnection = ((MySQL_CloudConnection)
sdkConnection).getMySQLConnection();
```

```
        try (Statement ps = mySQLConnection.createStatement()) {
            String dbForCustomQuery=((MySQL_CloudConnection)
sdkConnection).getCatalog();
            ps.executeQuery("USE "+dbForCustomQuery);
            ps.executeQuery(customQuery);
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            ExceptionManager.createNonNlsAdapterSDKException(
                    "An error occured while executing custom query for validation:["
+ e.getMessage() + "]");
            return false;
        }
        return true;
    }
```

Code Changes for Object Path Override Capability

When you enable the object path override capability for the connector, you must manually modify the Informatica Connector Toolkit classes and methods to implement the object path override logic.

Add the following code changes to implement the object path override capability for the connector:

1. Declare the following variable in the RuntimePlugin, RuntimeOperationAdapter, and RuntimeDataAdapter classes:

   ```
   private InfaUtils infaUtils;
   ```

2. Initialize the `infaUtils` variable in the initPlugin() method of the runtime plugin class.

   The following code snippet shows an example of how to initialize the `infaUtils` variable for MySQL_Cloud Connector:

   ```
   @Override
       public int initPlugin(PluginInfo pluginInfo, InfaUtils infaUtils){
               this.logger = infaUtils.getLogger();
               this.infaUtils=infaUtils;
               pluginInfo.setPluginDescription("");

   pluginInfo.setPluginName("com.infa.adapter.mysql_cloud.libraryInfo.TABLEMYSQL_CLOUDLI
   BRARYINFO");
               pluginInfo.setPluginVersion("1.0.0");
               return EStatus.SUCCESS;
       }
   ```

3. Add the `infaUtils` variable as a constructor argument in the RuntimeOperationAdapter and RuntimeDataAdapter classes.

   The following code snippet shows how to add the `infaUtils` variable as a constructor argument for MySQL_Cloud Connector:

   ```
   public MySQL_CloudTableOperationAdapter(Logger infaLogger,InfaUtils infaUtils ){
            this.logger = infaLogger;
            this.infaUtils=infaUtils;
       }

           MySQL_CloudTableDataAdapter(InfaUtils infaUtils){
           this.infaUtils=infaUtils;

       }
   ```

4. Update the constructors for the operation adapter and data adapter objects in the getDataAdapter() and getDataSourceOperationAdapter methods in the RuntimePlugin class.

The following code snippet shows an example for MySQL_Cloud Connector:

```
@Override
    public DataAdapter getDataAdapter(){
        return new MySQL_CloudTableDataAdapter(infaUtils);
    }

 @Override
    public DataSourceOperationAdapter getDataSourceOperationAdapter(){
        return new MySQL_CloudTableOperationAdapter(logger,infaUtils);
    }
```

5. Add an import statement in the RuntimeOperationAdapter class to access the `infaUtils` variable:

```
import com.informatica.sdk.adapter.javasdk.common.InfaUtils;
```

6. Access the value of the object path overridden parameters in the initDataSourceOperation() method of the RuntimeOperationAdapter class:

```
objectPathWrite =this.infaUtils.getObjectParameters();

//objectPathWrite is a Map with Native name of the object as key and overriden
parameter value as the value.
//Check if the overriden value is valid.
```

Code Changes for Pushdown Capability

When you enable full pushdown capability for the connector, you must manually modify the Informatica Connector Toolkit classes and methods to implement the pushdown logic.

The connector must validate if pushdown is supported for the extensionID of the source and generate the pushdown SQL query. Set the pushdown SQL query in the **PushdownRuntimeMetadata** object and validate if the SQL query is generated successfully. If the validation is unsuccessful, the mapping runs without full pushdown.

Add the following code changes to the **ASOOperationObjMgr.ValidateAll()** and **OperationAdapter.initDataSourceOperation()** method in the runtime.semantic and runtime.adapter package respectively:

1. In the ASOOperationObjMgr.ValidateAll() method in the runtime.semantic package, get the PushdownRuntimeMetadata object:

```
ASOOperation targetASO = (ASOOperation)currentObj;
PushdownRuntimeMetadata pushdownMetadata = targetASO.getASOPushdownRuntimeMetadata();
```

2. Validate if pushdown is supported for the source:

```
List<ASOOperation> sourceAso = pushdownMetadata.getAsoOperationList();
//Get a list of native sources and check for a source with extensionID other than
MySQL
for(ASOOperation aso:sourceAso){
String extentionID =
((SD_ASOOperation)aso)._get_imfObject().getBaseASO().getExtensionId();
if(!extentionID.equalsIgnoreCase("com.infa.adapter.mysql_cloud")){
Utils.createPushdownValidationError(currentObj,"Ecosystem Pushdown is not supported
for Combination of connectors MySQL and "+extentionID);
return false;
}
}
```

3. Create a target catalog to add all the source objects:

```
//Get the NMOTypeContext object to create a catalog & add records

NMOTypeContext SourceNMO= new
```

```
NMOTypeContext(pushdownMetadata.getAsoOperationList().get(0).getASO().getNmoType(),"P
rovide Target AdapterID);

//Get the Catalog to add the Source/Lookup records

Catalog catalog = SDKPushdownUtils.getCatalog(SourceNMO);
```

4. Get the list of OperationBase from the projection for the source objects

```
List<OperationBase> mergeOperation = new ArrayList<>();
List<OperationBase> baseOperations =
pushdownMetadata.getAsoOperationList().get(0).getReadProjection().getBaseOperations(O
perationTypeEnum.JOIN);
if (baseOperations != null && baseOperations.size() > 0) {
List<OperationBase> ops =
pushdownMetadata.getAsoOperationList().get(0).getReadProjection().getBaseOperations(O
perationTypeEnum.CONVERSION);
for (OperationBase op : ops) {
        mergeOperation.add(op);
        }
   }
else{
     List<OperationBase> opsNJ =
pushdownMetadata.getAsoOperationList().get(0).getReadProjection().getBaseOperations(O
perationTypeEnum.CONVERSION);
     for (OperationBase op : opsNJ) {
            mergeOperation.add(op);
            }
    }
```

5. Create flat records for all the source objects and add them to the list of source records:

```
//List of records for source
List<Record> listOfRecords=new ArrayList<Record>();
for(OperationBase operationBase : mergeOperation) {
            Factory sdkFactory = catalog.getFactory();
            FlatRecord flatRecord = sdkFactory.newFlatRecord(catalog);
            Operation operation = (Operation) operationBase;
            flatRecord.setName(tableName);
            flatRecord.setNativeName(tableName);

            List<FieldBase> fieldList =  (((SD_ConversionOperation)
operation).getOperationFields());
            //Populate the Flat record fields created for this source
            for(FieldBase  fld:fieldList)
            {
               Field field = sdkFactory.newField(catalog);
                  if(fld instanceof SAD_DerivedField) {
                      SD_DerivedField dField=(SD_DerivedField)fld;
               //Field nativeFld = (Field)fld;
               field.setName(dField.getName());
               field.setNativeName(dField.getName());
               field.setDataType(dField.getTypeName());
               field.setPrecision(dField.getPrecision());
               field.setScale(dField.getScale());
               flatRecord.addField(field);
             }
            }
            catalog.addRootRecord(flatRecord);
            //Add it to the list of source records
            listOfRecords.add(flatRecord)
}
```

6. Get the list of OperationBase from the projection for lookup objects:

```
List<OperationBase> lookupOpBase = new ArrayList<OperationBase>();
List<ASOOperation> asoOps = pushdownMetadata.getLookupASOOperationList();
if (asoOps != null && !asoOps.isEmpty()) {
for (ASOOperation eachASOOp : asoOps) {
List<OperationBase> ops =
eachASOOp.getReadProjection().getBaseOperations(OperationTypeEnum.CONVERSION);
lookupOpBase.add(ops.get(0));
```

```
}
}
```

7. Create flat records for all the lookup source objects and add them to the list of lookup records:

```
List<Record> lookupRecord=new ArrayList<Record>();
for(OperationBase operationBase : lookupOpBase) {
            Factory sdkFactory = catalog.getFactory();
            FlatRecord flatRecord = sdkFactory.newFlatRecord(catalog);
            Operation operation = (Operation) operationBase;
            flatRecord.setName(tableName);
            flatRecord.setNativeName(tableName);

            List<FieldBase> fieldList =  (((SD_ConversionOperation)
operation).getOperationFields());
            //Populate the Flat record fields created for this source
            for(FieldBase  fld:fieldList)
            {
               Field field = sdkFactory.newField(catalog);
                  if(fld instanceof SAD_DerivedField) {
                      SD_DerivedField dField=(SD_DerivedField)fld;
               //Field nativeFld = (Field)fld;
               field.setName(dField.getName());
               field.setNativeName(dField.getName());
               field.setDataType(dField.getTypeName());
               field.setPrecision(dField.getPrecision());
               field.setScale(dField.getScale());
               flatRecord.addField(field);
             }
            }
            catalog.addRootRecord(flatRecord);
            //Add it to the list of source records
            lookupRecord.add(flatRecord)
}
```

8. Create a PushdownUtilsContext object that provides all the connector instances needed to generate the pushdown SQL query:

```
//MySQL_Cloud is an example of the engine object to be passed as an argument for
creating PushdownUtilsContext

MySQL_CloudEngine mySQLEng = new MySQL_CloudEngine(new
MySQL_CloudASOOperationObjMgr());

PushdownUtilsContext pdctx=null;

//Create a new PushdownUtilsContext object based on whether a lookup source is used
the mapping

if(lookupOpBase!=null&&!lookupRecord.isEmpty()) {
pdctx= new PushdownUtilsContext(mySQLEng,
pushdownMetadata.getAsoOperationList().get(0).getReadProjection(),
((ASOOperation)currentObj).getWriteProjection(), ctx, mergeOperation, listOfRecords,
lookupOpBase, lookupRecord);
}
else {
pdctx= new PushdownUtilsContext(mySQLEng,
pushdownMetadata.getAsoOperationList().get(0).getReadProjection(),
((ASOOperation)currentObj).getWriteProjection(), ctx, mergeOperation,
listOfRecords);
}
```

9. Get the pushdown SQL query using the PushdownUtilsContext, optimize the pushdown SQL query, and set the optimized pushdown SQL query in the PushdownRuntimeMetadata object:

```
try
{
PushdownSql pdoSQL=SDKPushdownUtils.getPushdownSDKSQL(pdctx);
String finalOptmizedQuery=(String)pdoSQL.getOptimizedSql().get(0);
pushdownMetadata.setSql(finalOptmizedQuery);
if(finalOptmizedQuery.isEmpty()||finalOptmizedQuery==null) {
```

```
Utils.createPushdownValidationError(currentObj,"Pushdown SQL is empty");
return false;
}
}
} catch (ExpressionParseException e) {`
Utils.createPushdownValidationError(currentObj,e.getMessage());
return false;
}
Utils.logMessage(currentObj,"Full Pushdown Query is : "+pushdownMetadata.getSql());
Utils.logMessage(currentObj,"Pushdown Enabled Successfully");
return true;
```

10. Add any queries to create staging tables to custom metadata of PushdownRuntimeMetadata object. Create two lists createStagingTableQuery & dropStagingTableQuery when the list of the source or lookup records are added to the target catalog. You can add the queries in both the lists to the custom metadata of PushdownRuntimeMetadata to be accessed at runtime.

    Note: Perform Step 10 before you return the status of SQL query generation in step 9.

    ```
    for (int i = 0; i < this.createStagingTableQuery.size(); i++) {
    pushdownMetadata.addCustomMetadata(pushdownMetadata.getCustomMetadataSize(),
    (String)this.createStagingTableQuery.get(i));
    }
    /**Populate the query to create staging tables in custom metadata of
    PushdownRuntimeMetadataObject
    * The custom metadata can be accessed at runtime to execute these queries
    */
    pushdownMetadata.addCustomMetadata(pushdownMetadata.getCustomMetadataSize(),
    "Partition");
    for (int j = 0; j < this.dropStagingTableQuery.size(); j++) {
    pushdownMetadata.addCustomMetadata(pushdownMetadata.getCustomMetadataSize(),
    (String)this.dropStagingTableQuery.get(j));
    }
    ```

11. Access the source connection attributes and read runtime attributes.

    Note: Perform Step 11 before returning the status of SQL query generation in step 9.

    The following code snippet shows an example for MySQL_Cloud Connector:

    ```
    private void getSourceConnectionAndReadAttrs(PushdownRuntimeMetadata
    pushdownMetadata, MetadataObject currentObj)
    {

    List<OperationBase>
    opsNativeSource=pushdownMetadata.getAsoOperationList().get(0).getReadProjection().get
    BaseOperations(OperationTypeEnum.NATIVE_SOURCE);
        for (OperationBase op : opsNativeSource) {
            Map<String, Object>
    srcConn=SDKPushdownUtils.getNativeSourceConnection(pushdownMetadata.getAsoOperationLi
    st().get(0).getReadProjection(),(NativeSource) op);
            String nativeSourceName=((NativeSource)
    op).getNativeRecord().getNativeName();
            String nativeSourceCatalog=(String) srcConn.get("catalog");
            String nativeSourceHost=(String) srcConn.get("host");
            Utils.logMessage(currentObj,"Connection Attributes for Source
    "+nativeSourceName);
            Utils.logMessage(currentObj,"Catalog: "+nativeSourceCatalog+"
    Host:"+nativeSourceHost);
            Map<String, Object>
    nativeSourceReadAttrs=SDKPushdownUtils.getNativeSourceReadAttributes(pushdownMetadata
    .getAsoOperationList().get(0).getReadProjection(), (NativeSource) op);
            Utils.logMessage(currentObj,"Read Runtime Attributes for Source
    "+nativeSourceName);
            String nativeSourcePreSQL=(String) nativeSourceReadAttrs.get("preSQL");
            String nativeSourcePostSQL=(String) nativeSourceReadAttrs.get("postSQL");
            int nativeSourceRowOffSet=  (int) nativeSourceReadAttrs.get("rowOffSet");
            int nativeSourceRowLimit= (int) nativeSourceReadAttrs.get("rowLimit");
            long nativeSourceThresholdLimit=(long)
    nativeSourceReadAttrs.get("thresholdLimit");
            Utils.logMessage(currentObj,"PreSQL: "+nativeSourcePreSQL);
    ```

```
            Utils.logMessage(currentObj,"PostSQL: "+nativeSourcePostSQL);
            Utils.logMessage(currentObj,"RowOffSet: "+nativeSourceRowOffSet);
            Utils.logMessage(currentObj,"RowLimit: "+nativeSourceRowLimit);
            Utils.logMessage(currentObj,"RowLimit: "+nativeSourceThresholdLimit);
           }

        }
```

12. In the OperationAdapter.initDataSourceOperation() method in the runtime.adapter package, if the DataSourceOperation object is an instance of the write capability and PushdownRuntimeMetadata object returns an SQL query, process the pushdown SQL query returned by the PushdownRuntimeMetadata.getSql(). You must drop the staging tables and close any open connections before returning the status.

    The following code snippet shows an example for MySQL_Cloud Connector:

```
//Pushdown query execution for MySQL_Cloud is given below
PushdownRuntimeMetadata pushdownMeta =
dsoHandle.getAdapterDataSourceOperation().getASOPushdownRuntimeMetadata();
if (pushdownMeta != null)
              {
                int j=0;
                for(int i=0;i<pushdownMeta.getCustomMetadataSize();i++){
                String createStagingTableQuery = pushdownMeta.getCustomMetadata(i);
                if(createStagingTableQuery.equalsIgnoreCase("partition")){
                    j=i;
                    break;
                }
                writeLog(EMessageLevel.MSG_INFO, "Create Staging Table for PDO with
Query:" + createStagingTableQuery);
                tstmt.executeUpdate(createStagingTableQuery);
                }
                writeLog(EMessageLevel.MSG_INFO, "Full PDO Query is:" +
pushdownMeta.getSql());
                rowInserted=tstmt.executeUpdate(pushdownMeta.getSql());
                for(int k=j+1;k<pushdownMeta.getCustomMetadataSize();k++){
                writeLog(EMessageLevel.MSG_INFO, "Dropping Staging Table for PDO
with Query:" + pushdownMeta.getCustomMetadata(k));
                tstmt.executeUpdate(pushdownMeta.getCustomMetadata(k));
                }
                if(rowInserted > 0){
                    //Set the RowStats for executed PDO Query
                    RowsStatInfo rowstatInfoPDO =
runtimeConfigHandleInit.getRowsStatInfo(EIUDIndicator.INSERT);
                    rowstatInfoPDO.incrementRequested(rowInserted);
                    rowstatInfoPDO.incrementAffected(rowInserted);
                    rowstatInfoPDO.incrementApplied(rowInserted);
                }

              }
            // closing statements and connection to data source
        } catch (Exception e) {
                writeLog(EMessageLevel.MSG_ERROR, e1.getMessage());
                return EStatus.FAILURE;
            }
```

13. In the ASOOperationObjMgr.ValidateAll() method in the runtime.semantic package, override the TypeHandler, QueryVisitor, Engine, and PostProcessor methods to generate and validate the pushdown SQL query.

    The following code snippet shows an example for MySQL_Cloud Connector:

```
public MySQL_CloudTypeHandler getTypeHandler()
    {
        return new MySQL_CloudTypeHandler();
    }

    /**
     * connector specific QueryVisitor instance should be returned here by connector
specific implementation of MD_ASOOperation
```

```
**/
@Override
public  XVisitor<SdkXNode, SDKExprNode> getSDKXVisitor(XVisitorContext ctx) {
    return new MySQL_CloudXVisitor<SdkXNode>(ctx.getLog(), ctx.getScope());
}

@Override
public SDKPostProcessor getQueryPostProcessor(PostProcessorContext ctx)
{
    return new MySQL_CloudPostProc(ctx);
}

/**
 * Get connector specific Engine (used by CDI-e pushdown framework currently)
 * @return connector specific engine
 */
public  SDKEngine getAdapterEngine(SDKEngineContext ctx)
{
    return new MySQL_CloudEngine();
}
```

Code Changes for SQL Transformation

When you create a endpoint metadata object for procedure pattern using an SQL transformation, you must manually modify the Informatica Connector Toolkit classes and methods to implement SQL transformation.

The changes are auto-generated if the code generation for runtime is executed for the first time or the first endpoint metadata object is being created. If the endpoint metadata object created is not the first one, perform the following steps:

1. Add the following import statement in the RuntimeDataAdapter class:

```
import
com.informatica.sdk.adapter.metadata.common.typesystem.typelibrary.semantic.iface.Str
ucturalFeature;
import
com.informatica.sdk.adapter.metadata.projection.sinkoperation.semantic.iface.Platform
Sink;
import
com.informatica.sdk.adapter.metadata.patternblocks.procedure.semantic.iface.Procedure
;
```

2. Declare the variables in the RuntimeDataAdapter class as shown below:

```
private String procName=null;
private String sqlTxQuery = null;
private List<StructuralFeature> sfList = null;
private Procedure pr = null;
```

3. Use the following code in the initDataSession method of RuntimeDataAdapter class to initialize the variables in step 2.

```
List<Capability> caps1 = m_asoOperation1.getCapabilities();
        Capability cap1 = caps1.get(0);
        if (cap1 instanceof CallCapability)
        {
            Projection readProj = m_asoOperation1.getReadProjection();
            NativeSource nativeSrcOp = (NativeSource)
(readProj.getBaseOperations(OperationTypeEnum.NATIVE_SOURCE)
                    .get(0));
            Procedure m_nativeProcedure = (Procedure)
((nativeSrcOp).getNativeRecord());
            PlatformSink platformSink = (PlatformSink)
(readProj.getBaseOperations(OperationTypeEnum.PLATFORM_SINK).get(0));
```

```
            this.sfList = platformSink.getComplexType().getStructuralFeatures();
            this.pr=m_nativeProcedure;
            this.procName = m_nativeProcedure.getName();
                        this.sqlTxQuery = pr.getQuery();
            return EReturnStatus.SUCCESS;
        }
```

4. Implement the call method in the RuntimeDataAdapter class.

   The following code snippet shows an example for MySQL_Cloud Connector:

```
@Override
    public int call(DataSession dataSession, CallAttributes callAttr) throws
SDKException
    {
        InfaUtils pInfaUtils = dataSession.getInfaUtilsHandle();
        logger = pInfaUtils.getLogger();
        MySQL_CloudTableDataConnection conn = (MySQL_CloudTableDataConnection)
dataSession.getConnection();
        Connection nativeConn = (Connection) conn.getNativeConnection();
        int paramListSize = pr.getFieldList().size();
        if(this.sqlTxQuery!=null && !this.sqlTxQuery.isEmpty())
        {
            return executeAdhocSQLQuery(dataSession,callAttr,nativeConn);
        }
        return EReturnStatus.NO_MORE_DATA;
    }
```

5. Use the following sample code for the method executeAdhocSQLQuery referenced in step 4:

```
/**
    * The SQL Query provided by the user in SQL Tx is executed once for each input.
    * SQL Query provided by the user can be static or dynamic. Dynamic queries will
contain parameters.
    * Allowed syntax for parameters are ~tablename~ for tablenames and ?columname?
for columnnames. Users can provide a input port from the source as a parameter.
Connector needs to replace the values for these parameters before query execution.
    * SQLError is always the first field in SQL Tx. SQLError field is null/empty if
query is executed successfully.
    * If user selects RowsEffected checkbox in SQL Tx, then RowsEffected will be the
second field in SQL Tx. Connector developer should populate this field for every row
based on the number of rows effected by the query executed.
    * If SQL Tx is set as a Active transformation, then connector developer needs to
return only one row for each query executed against the input row from the source.
    * MAX_OUT_ROW_COUNT indicates the maximum number of rows connector should return
for each query.
    * If MAX_OUT_ROW_COUNT is 0, then there is no limit on maximum number of rows
connector should return for each query.
    * Incase of error in query execution, connector developer should return one row
with all null values and the error message in the SQLError field.
    * Connector should return NO_MORE_DATA if all the rows are read, else return
SUCCESS
    **/

    public int executeAdhocSQLQuery(DataSession dataSession, CallAttributes
callAttr, Connection nativeConn){
    int returnStatus;
    String query = this.sqlTxQuery;
     /**
       * Access the SQL Tx properties
       * String isActiveTransformation=this.infaUtils.getExecEnvStringProperty("IS
ACTIVE");
       * String
maxOutRowCount=this.infaUtils.getExecEnvStringProperty("MAX_OUT_ROW_COUNT");
     **/
     int  outputBufferCapacity=callAttr.getOutputBufCapacity();
    //replace any parameters used in the query with values
    for(int i=0; i < this.inputParamList.size(); i++) {
        String paramDataType =this.inputParamList.get(i).getDataType();
        String paramName = this.inputParamList.get(i).getName();
        DataAttributes dataAttr = new DataAttributes();
```

```
        dataAttr.setRowIndex(0);
        dataAttr.setColumnIndex(i);
        //adapter can read their respective types from data session
        String data;
        try {
            data = dataSession.getStringData(dataAttr);
            } catch (SDKException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                return EReturnStatus.FAILURE;
                }
                query=getResolvedAdhocQuery(query,paramName,data);
        }
        //Execute the query
        //rs = st.executeQuery(query);
        List<List<Object>> result = new ArrayList<List<Object>>();
        returnStatus = readDatafromAdhocQuery(dataSession, result,
outputBufferCapacity);
                try {
                    setProcDataToPlatform(dataSession, result);
                } catch (SDKException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                    return EReturnStatus.FAILURE;
                }
            return returnStatus;
    }
```

6. Use the following sample code for the getResolvedAdhocQuery method referenced in step 5:

```
/**
         * replaces the parameters in the query with data values from the input ports
         * @param query
         * @param paramName
         * @param data
         * @return
         */

    public String getResolvedAdhocQuery(String query, String paramName, String data){
        //Sample code to replace parameters in the sql query with values
        String param1="~"+ paramName+"~";
        String param2="~ "+ paramName+ " ~";
        String param3="?"+ paramName + "?";
        String param4="? "+ paramName + " ?";
        if(query.indexOf(param1) != -1) {
            query = query.replace(param1, data);
        }
        if(query.indexOf(param2) != -1) {
            query = query.replace(param2, data);
        }
        if(query.indexOf(param3) != -1) {
            data="'"+data+"'";
            query = query.replace(param3, data);
        }
        if(query.indexOf(param4) != -1) {
            data="'"+data+"'";
            query = query.replace(param4, data);
        }
            return query;
    }
```

7. Use the following sample code for the readDatafromAdhocQuery method referenced in step 5:

```
public int readDatafromAdhocQuery(DataSession dataSession, List<List<Object>>
result, int outputBufferCapacity){

        //    Example code to fetch data from the jdbc result set
        /*    int rowsRead = 0;
            try {
                while (rs.next() && rowsRead++ < outputBufferCapacity) {
                    List<Object> datarow = new ArrayList<Object>();
```

```
                    for (int i = 0; i < connectedFields.size(); i++) {
                    //adapter can read their respective datatypes instead of object
                        datarow.add(rs.getObject(i+1));
                    }
                    dataTable.add(datarow);
                }
            } catch (SQLException e) {
                logger.logMessage(EMessageLevel.MSG_FATAL_ERROR,
ELogLevel.TRACE_NONE, e.getMessage());
                return EReturnStatus.FAILURE;
            }
            if (rowsRead == outputBufferCapacity)
                return EReturnStatus.SUCCESS;
        */
        // TODO: Replace the sample code with native code to read data from the
native object
        // Sample dummy code to populate a projected source with one column of
string type and two rows
        List<Object> datarow = new ArrayList<Object>();
        datarow.add("Stan");
        List<Object> datarow2 = new ArrayList<Object>();
        datarow2.add("Mark");
        result.add(datarow);
        result.add(datarow2);
        return EReturnStatus.NO_MORE_DATA;
    }
```

8. Use the following sample code for the method setProcDataToPlatform referenced in step 5:

```
    /**
         * Sets the multiple row data in the data table to the data session buffer
         *
         * <pre>
         * ##################################
         * AUTOGENERATED CODE
         * ##################################
         * </pre>
         *
         * @param dataSession
         *            The dataSession instance, which is the container for SDK
         *            handles.
         * @param dataTable
         *            List of List of Objects. Each List of Objects represents a
         *            single row.
         */

    public void setProcDataToPlatform(DataSession dataSession, List<List<Object>>
dataTable) throws SDKException  {
    for (int row = 0; row < dataTable.size(); row++) {
                List<Object> rowData = dataTable.get(row);
                for (int col = 0; col <  dataTable.get(0).size(); col++) {
                    DataAttributes pDataAttributes = new DataAttributes();
                    pDataAttributes.setDataSetId(0);
                    pDataAttributes.setColumnIndex(col);
                    pDataAttributes.setRowIndex(row);
                    Object data = rowData.get(col);

                    String dataType = sfList.get(col).getDataType();
                    String columnName = sfList.get(col).getName();

                    if (dataType.equalsIgnoreCase("string") ||
dataType.equalsIgnoreCase("text")) {
                        if (data == null) {
                            pDataAttributes.setLength(0);
                            dataSession.setStringData((String) data,
pDataAttributes);
                        } else {
                            String text = data.toString();

                            int fieldLength = sfList.get(col).getPrecision();
                            if (text.length() > fieldLength) {
```

```
                                pDataAttributes.setLength(fieldLength);
                                pDataAttributes.setIndicator(EIndicator.TRUNCATED);
                                dataSession.setStringData(text.substring(0,
fieldLength), pDataAttributes);
                            } else {
                                pDataAttributes.setLength(text.length());
                                pDataAttributes.setIndicator(EIndicator.VALID);
                            }
                            dataSession.setStringData(text, pDataAttributes);
                        }
                    } else if (dataType.compareToIgnoreCase("double") == 0) {
                        if (data instanceof Double) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).doubleValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setDoubleData((Double) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("float") == 0) {
                        if (data instanceof Float) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).floatValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setFloatData((Float) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("long") == 0) {
                        if (data instanceof Long) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).longValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setLongData((Long) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("short") == 0) {
                        if (data instanceof Short)
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).shortValue();
```

```
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setShortData((Short) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("integer") == 0) {
                        if (data instanceof Integer) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).intValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setIntData((Integer) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("bigint") == 0) {
                        if (data instanceof Long) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).longValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setLongData((Long) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("date/time") == 0) {
                        if (data instanceof Timestamp) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Date) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            Timestamp ts = new Timestamp(((Date) data).getTime());
                            data = ts;
                        } else if (data instanceof Time) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            Timestamp ts = new Timestamp(((Time) data).getTime());
                            data = ts;
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "]"
                                            + " should be a of type [" +
Timestamp.class.getName() + "].");
                            data = null;
                        }
```

```
                        dataSession.setDateTimeData((Timestamp) data,
pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("binary") == 0) {
                        if (data instanceof byte[]) {
                            pDataAttributes.setLength(((byte[]) data).length);
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_DEBUG,
ELogLevel.TRACE_VERBOSE_DATA, "Data for type ["
                                    + dataType + "] should be a of type [" +
byte[].class.getName() + "].");
                            data = null;
                        }
                        dataSession.setBinaryData((byte[]) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("decimal") == 0) {
                        if (data instanceof BigDecimal) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_DEBUG,
ELogLevel.TRACE_VERBOSE_DATA, "Data for type ["
                                    + dataType + "] should be a of type [" +
BigDecimal.class.getName() + "].");
                            data = null;
                        }
                        dataSession.setBigDecimalData((BigDecimal) data,
pDataAttributes);
                    }
                }
            }
        }
```

Code Changes for Stored Procedures

When you create a endpoint metadata object for procedure pattern using stored procedures, you must manually modify the Informatica Connector Toolkit classes and methods to implement stored procedures.

The changes are auto-generated when you run the runtime code for the first time ot when you create the first endpoint metadata object. If the endpoint metadata object created is not the first one, perform the following steps:

1. Add the following import statement in the RuntimeDataAdapter class:

```
import
com.informatica.sdk.adapter.metadata.common.typesystem.typelibrary.semantic.iface.Str
ucturalFeature;
import
com.informatica.sdk.adapter.metadata.projection.sinkoperation.semantic.iface.Platform
Sink;
import
com.informatica.sdk.adapter.metadata.patternblocks.procedure.semantic.iface.Procedure
;
```

2. Declare the following variables in the RuntimeDataAdapter class:

```
private String procName=null;
private List<StructuralFeature> sfList = null;
private Procedure pr = null;
private ArrayList<Parameter> inputParamList = null;
private ArrayList<Parameter> outputParamList = null;
```

3. Use the following code in the initDataSession method of RuntimeDataAdapter class to initialize the variables in step 2.

```
                List<Capability> caps1 = m_asoOperation1.getCapabilities();
        Capability cap1 = caps1.get(0);
        if (cap1 instanceof CallCapability) {
            Projection readProj = m_asoOperation1.getReadProjection();
            NativeSource nativeSrcOp = (NativeSource)
(readProj.getBaseOperations(OperationTypeEnum.NATIVE_SOURCE)
                    .get(0));
            Procedure m_nativeProcedure = (Procedure)
((nativeSrcOp).getNativeRecord());
            PlatformSink platformSink = (PlatformSink)
(readProj.getBaseOperations(OperationTypeEnum.PLATFORM_SINK).get(0));
            this.sfList = platformSink.getComplexType().getStructuralFeatures();
            this.pr=m_nativeProcedure;
            this.procName = m_nativeProcedure.getName();
            this.inputParamList = new ArrayList<Parameter>();
            this.outputParamList = new ArrayList<Parameter>();

List<com.informatica.sdk.adapter.metadata.field.semantic.iface.FieldBase> paramList
= m_nativeProcedure.getFieldList();
            for (com.informatica.sdk.adapter.metadata.field.semantic.iface.FieldBase
fieldBase : paramList) {
                Parameter param = (Parameter)fieldBase;
                ParameterTypeEnum type = param.getParameterTypeEnum();
                switch(type) {
                    case INOUT_TYPE:
                        this.inputParamList.add(param);
                        this.outputParamList.add(param);
                        break;
                    case IN_TYPE:
                        this.inputParamList.add(param);
                        break;
                    case OUT_TYPE:
                        this.outputParamList.add(param);
                        break;
                    default:
                }
            }
            return EReturnStatus.SUCCESS;
        }
```

4. Implement the call method in the RuntimeDataAdapter class.

   The following code snippet shows an example on how to implement the call method for MySQL_Cloud Connector:

```
@Override
    public int call(DataSession dataSession, CallAttributes callAttr) throws
SDKException{
        InfaUtils pInfaUtils = dataSession.getInfaUtilsHandle();
        logger = pInfaUtils.getLogger();
        MySQL_CloudTableDataConnection conn = (MySQL_CloudTableDataConnection)
dataSession.getConnection();
        Connection nativeConn = (Connection) conn.getNativeConnection();
        int paramListSize = pr.getFieldList().size();
        String procQuery = "{CALL " +  procName + "(";
        for (int i=0 ; i < paramListSize; i++) {
            if(i+1==paramListSize) {
                procQuery=procQuery+ "?";
                break;
            }
            procQuery=procQuery+ "?,";

        }
        procQuery = procQuery + ")}";
        CallableStatement callStmt = null;
        int col = 0;
        int rowSize = callAttr.getNumInputRows();
```

```
        List<List<Object>> result = new ArrayList<List<Object>>();
        String stringValue = null;
        BigDecimal bdc=null;
        Timestamp ts=null;
        Date dt=null;
        Time time=null;
        double doubleValue;
        float floatValue;
        long longValue;
        int intValue;
        try {
            for(int i=0; i < rowSize; i++) {
                col=0;
                callStmt = nativeConn.prepareCall(procQuery);
                for
(com.informatica.sdk.adapter.metadata.field.semantic.iface.FieldBase fieldBase : pr
                        .getFieldList()) {
                    Parameter param = (Parameter) fieldBase;
                    String dataType = param.getDataType();
                    DataAttributes dataAttr = new DataAttributes();
                        switch (dataType) {
                        case "TINYINT"  :
                        case "SMALLINT" :
                        case "MEDIUMINT":
                        case "INT":

if(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                stringValue = dataSession.getStringData(dataAttr);
                                if(stringValue==null)
                                {
                                    callStmt.setNull(col + 1,Types.INTEGER);
                                }
                                else {
                                intValue = Integer.parseInt(stringValue);
                                callStmt.setInt(col + 1, intValue);
                                }
                            }
                            else {
                                if(dataType.equalsIgnoreCase("TINYINT")) {
                                    callStmt.registerOutParameter(col+1,
Types.TINYINT);
                                }
                                else if(dataType.equalsIgnoreCase("SMALLINT")) {
                                callStmt.registerOutParameter(col+1, Types.SMALLINT);
                                }
                                else {
                                    callStmt.registerOutParameter(col+1,
Types.INTEGER);
                                }
                            }

                            break;
                        case "BIGINT":

if(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                stringValue = dataSession.getStringData(dataAttr);
                                if(stringValue==null)
                                {
                                    callStmt.setNull(col + 1,Types.BIGINT);
                                }
                                else {
                                     longValue = Long.parseLong(stringValue);
                                        callStmt.setLong(col + 1, longValue);
```

```
                                }
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.BIGINT);
                            }
                            break;

                        case "DOUBLE":

if(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                stringValue = dataSession.getStringData(dataAttr);
                                if(stringValue==null)
                                {
                                    callStmt.setNull(col + 1,Types.DOUBLE);
                                }
                                else {
                                    doubleValue = Double.parseDouble(stringValue);
                                    callStmt.setDouble(col + 1, doubleValue);
                                }
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.BIGINT);
                            }
                            break;

                        case "FLOAT":

if(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                stringValue = dataSession.getStringData(dataAttr);
                                if(stringValue==null)
                                {
                                    callStmt.setNull(col + 1,Types.FLOAT);
                                }
                                else {
                                      floatValue = Float.parseFloat(stringValue);
                                      callStmt.setFloat(col + 1, floatValue);
                                }
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.BIGINT);
                            }
                            break;

                        case "BIT":
                            if
(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                stringValue = dataSession.getStringData(dataAttr);
                                if(stringValue.equalsIgnoreCase("true"))
                                    stringValue="1";
                                else
                                    stringValue="0";
                                if(stringValue==null) {
                                    callStmt.setNull(col+1, Types.BIT);
                                }
                                else
                                callStmt.setString(col+1, stringValue);
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.BIT);
                            }
```

```
                            break;
                        case "CHAR":
                            if
(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                stringValue = dataSession.getStringData(dataAttr);
                                if(stringValue==null) {
                                    callStmt.setNull(col+1, Types.CHAR);
                                }
                                else
                                callStmt.setString(col+1, stringValue);
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.CHAR);
                            }
                            break;

                        case "VARCHAR":
                            if
(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                stringValue = dataSession.getStringData(dataAttr);
                                if(stringValue==null) {
                                    callStmt.setNull(col+1, Types.VARCHAR);
                                }
                                else
                                callStmt.setString(col+1, stringValue);
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.VARCHAR);
                            }
                            break;

                        case "DATETIME":
                        case "TIMESTAMP":
                            if
(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                ts = dataSession.getDateTimeData(dataAttr);
                                if(ts==null) {
                                    callStmt.setNull(col+1, Types.TIMESTAMP);
                                }
                                else
                                 callStmt.setTimestamp(col+1,ts);
                            }
                            else {
                                callStmt.registerOutParameter(col+1,
Types.TIMESTAMP);
                            }

                            break;

                        case "DATE":
                            if
(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                ts = dataSession.getDateTimeData(dataAttr);
                                if(ts==null) {
                                    callStmt.setNull(col+1, Types.DATE);
                                }
                                else {
```

```
                                dt = new Date(ts.getTime());
                                callStmt.setDate(col+1,dt);
                                }
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.DATE);
                            }
                            break;

                        case "TIME":
                            if
(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                ts = dataSession.getDateTimeData(dataAttr);
                                if(ts==null) {
                                    callStmt.setNull(col+1, Types.TIME);
                                }
                                else {
                                 time = new Time(ts.getTime());
                                 callStmt.setTime(col+1,time);
                                }
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.TIME);
                            }
                            break;

                        case "DECIMAL":
                            if
(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                bdc=dataSession.getBigDecimalData(dataAttr);
                                if(bdc==null) {
                                    callStmt.setNull(col+1, Types.DECIMAL);
                                }
                                else
                                 callStmt.setBigDecimal(col+1,bdc);
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.DECIMAL);
                            }
                            break;

                        case "NUMERIC":
                            if
(param.getParameterTypeEnum()==ParameterTypeEnum.IN_TYPE ||
param.getParameterTypeEnum()==ParameterTypeEnum.INOUT_TYPE) {
                                dataAttr.setRowIndex(i);
                                dataAttr.setColumnIndex(col);
                                bdc=dataSession.getBigDecimalData(dataAttr);
                                if(bdc==null) {
                                    callStmt.setNull(col+1, Types.NUMERIC);
                                }
                                else
                                 callStmt.setBigDecimal(col+1,bdc);
                            }
                            else {
                                callStmt.registerOutParameter(col+1, Types.NUMERIC);
                            }
                            break;

                         default :
                               logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,"MySQL Stored Procedure encountered failure for the field "
+               param.getNativeName() + " with the value " +
```

```
dataSession.getStringData(dataAttr));
                                logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,"MySQL Stored Procedure is not supported for the datatype
"+dataType);
                                return EReturnStatus.FAILURE;
                            }

                        col++;
                    }

                    callStmt.executeQuery();
                    List<Object> resultRow = new ArrayList<Object>();
                    for (Parameter outParam : this.outputParamList) {
                        Object val=null;
                        switch(outParam.getDataType().toUpperCase()) {
                        case "TIMESTAMP":
                        val = callStmt.getTimestamp(outParam.getName());
                        resultRow.add(val);
                        break;

                        case "DATE":
                        val= callStmt.getDate(outParam.getName());
                        resultRow.add(val);
                        break;

                        case "TIME":
                        val = callStmt.getTime(outParam.getName());
                        resultRow.add(val);
                        break;

                        default:
                        val = callStmt.getObject(outParam.getName());
                        resultRow.add(val);
                        break;
                        }
                    }
                    result.add(resultRow);
                }
                setProcDataToPlatform(dataSession,result);
                callAttr.setNumRowsInOutputBuffer(result.size());
                if(callStmt!=null)
                    callStmt.close();
            } catch (SQLException e) {
                e.printStackTrace();
                return EReturnStatus.FAILURE;
            }
            return EReturnStatus.NO_MORE_DATA;
        }
```

5. Use the following sample code for the setProcDataToPlatform method referenced in step 4.

```
//**
         * Sets the multiple row data in the data table to the data session buffer
         *
         * <pre>
         * ###################################
         * AUTOGENERATED CODE
         * ###################################
         * </pre>
         *
         * @param dataSession
         *            The dataSession instance, which is the container for SDK
         *            handles.
         * @param dataTable
         *            List of List of Objects. Each List of Objects represents a
         *            single row.
         */

        public void setProcDataToPlatform(DataSession dataSession, List<List<Object>>
dataTable) throws SDKException  {
        for (int row = 0; row < dataTable.size(); row++) {
```

```
                List<Object> rowData = dataTable.get(row);
                for (int col = 0; col <  dataTable.get(0).size(); col++) {
                    DataAttributes pDataAttributes = new DataAttributes();
                    pDataAttributes.setDataSetId(0);
                    pDataAttributes.setColumnIndex(col);
                    pDataAttributes.setRowIndex(row);
                    Object data = rowData.get(col);

                    String dataType = sfList.get(col).getDataType();
                    String columnName = sfList.get(col).getName();

                    if (dataType.equalsIgnoreCase("string") ||
dataType.equalsIgnoreCase("text")) {
                        if (data == null) {
                            pDataAttributes.setLength(0);
                            dataSession.setStringData((String) data,
pDataAttributes);
                        } else {
                            String text = data.toString();

                            int fieldLength = sfList.get(col).getPrecision();
                            if (text.length() > fieldLength) {
                                pDataAttributes.setLength(fieldLength);
                                pDataAttributes.setIndicator(EIndicator.TRUNCATED);
                                dataSession.setStringData(text.substring(0,
fieldLength), pDataAttributes);
                            } else {
                                pDataAttributes.setLength(text.length());
                                pDataAttributes.setIndicator(EIndicator.VALID);
                            }
                            dataSession.setStringData(text, pDataAttributes);
                        }
                    } else if (dataType.compareToIgnoreCase("double") == 0) {
                        if (data instanceof Double) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).doubleValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setDoubleData((Double) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("float") == 0) {
                        if (data instanceof Float) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).floatValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setFloatData((Float) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("long") == 0) {
                        if (data instanceof Long) {
```

```
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).longValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setLongData((Long) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("short") == 0) {
                        if (data instanceof Short)
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).shortValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setShortData((Short) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("integer") == 0) {
                        if (data instanceof Integer) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).intValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setIntData((Integer) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("bigint") == 0) {
                        if (data instanceof Long) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Number) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            data = ((Number) data).longValue();
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "] "
                                            + "should be a of type [" +
Number.class.getName() + "] or its sub-types.");
                            data = null;
                        }
                        dataSession.setLongData((Long) data, pDataAttributes);
```

```
                    } else if (dataType.compareToIgnoreCase("date/time") == 0) {
                        if (data instanceof Timestamp) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data instanceof Date) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            Timestamp ts = new Timestamp(((Date) data).getTime());
                            data = ts;
                        } else if (data instanceof Time) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                            Timestamp ts = new Timestamp(((Time) data).getTime());
                            data = ts;
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_ERROR,
ELogLevel.TRACE_NONE,
                                    "Data for column [" + columnName + "] of type ["
+ dataType + "]"
                                            + " should be a of type [" +
Timestamp.class.getName() + "].");
                            data = null;
                        }
                        dataSession.setDateTimeData((Timestamp) data,
pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("binary") == 0) {
                        if (data instanceof byte[]) {
                            pDataAttributes.setLength(((byte[]) data).length);
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_DEBUG,
ELogLevel.TRACE_VERBOSE_DATA, "Data for type ["
                                    + dataType + "] should be a of type [" +
byte[].class.getName() + "].");
                            data = null;
                        }
                        dataSession.setBinaryData((byte[]) data, pDataAttributes);
                    } else if (dataType.compareToIgnoreCase("decimal") == 0) {
                        if (data instanceof BigDecimal) {
                            pDataAttributes.setIndicator(EIndicator.VALID);
                        } else if (data == null) {
                            pDataAttributes.setIndicator(EIndicator.NULL);
                        } else {
                            logger.logMessage(EMessageLevel.MSG_DEBUG,
ELogLevel.TRACE_VERBOSE_DATA, "Data for type ["
                                    + dataType + "] should be a of type [" +
BigDecimal.class.getName() + "].");
                            data = null;
                        }
                        dataSession.setBigDecimalData((BigDecimal) data,
pDataAttributes);
                    }
                }
            }
        }
```

CHAPTER 11

Run-time Behavior

This chapter includes the following topics:

Run-time Behavior Overview

Use the functions in the Informatica Connector Toolkit API to specify the run-time behavior of the connector. You must write the code to define how the connector connects, disconnects, reads from and writes to the data source.

The run-time functions are available in Java. Within the run-time functions, you can use any API that is appropriate for communicating with the data source. You must implement all the run-time functions.

The run-time functions use character string arguments and character data buffers in UCS-2 format. If the database API communicates with the data source through the UCS-2 character set, then pass the character strings and data buffers directly to the database API. If the database API does not use the UCS-2 character set, you must convert the data to UCS-2 format before you pass the data to the database API.

To reduce complexity, design the connector so that it does not require the end user to specify any character set information. Use the Unicode mode if the database client API provides a Unicode mode. Or, query the database to determine the correct character set to use when reading or writing data to the database. If you require input from the end user about the correct character set, define a custom connection attribute to store this information.

You can define the connector run-time behavior to support pre-SQL and post-SQL commands to perform tasks before and after a mapping run. For example, you can define the connector run-time behavior to support a pre command that initializes environment variables before the mapping run.

Run-time Java Functions

Extend the following classes to define the connector run-time behavior with Java functions:

DataConnection

Implement the methods in this class to connect and disconnect from the data source.

DataAdapter

Implement the methods in this class to initialize the data session, deinitialize the data session, begin the data session, end the data session, read data from the data source, and write data to the data source.

AutoPartitioningMetadataAdapter

If the connector supports partitioning capability, implement the methods in this class to specify the partition type and logic.

OperationAdapter

If the connector supports partitioning capability, implement the methods in this class to perform any operation before or after the partitioning.

ASOOperationObjMgr

Implement the methods in this class to perform any custom metadata validations.

CHAPTER 12

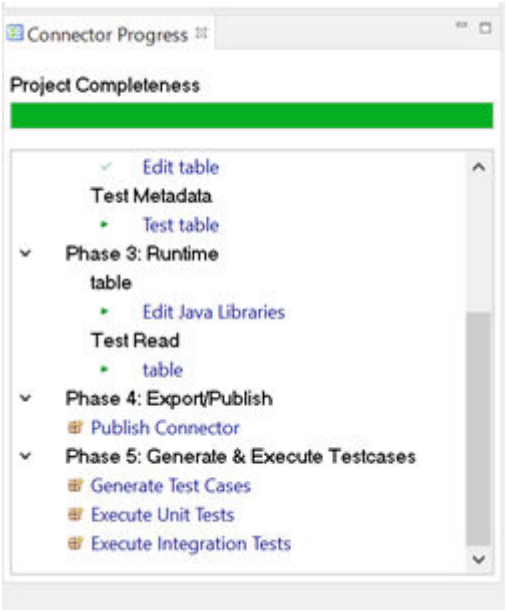
Test a Connector

This chapter includes the following topics:

Generating the Test Case for Unit and Integration Tests

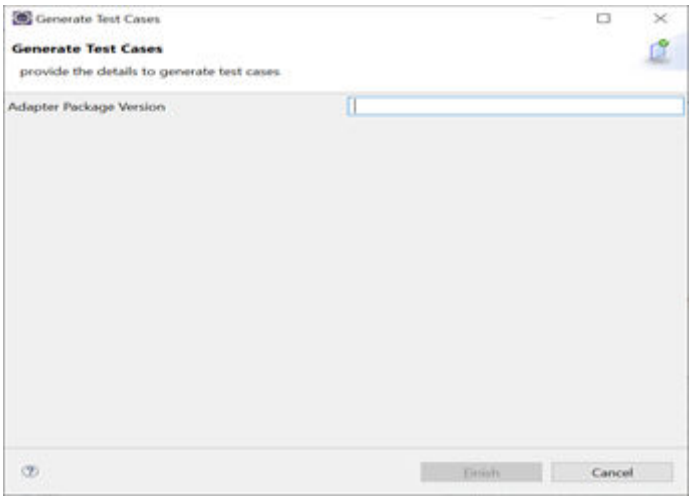
You can generate the test cases for unit and integration tests after you export the connector.

1. On the **Connector Progress** tab, select the **Generate Test Cases** option from the **Phase 5: Generate and Execute Test Cases** menu.



   The **Generate Test Cases** window appears.

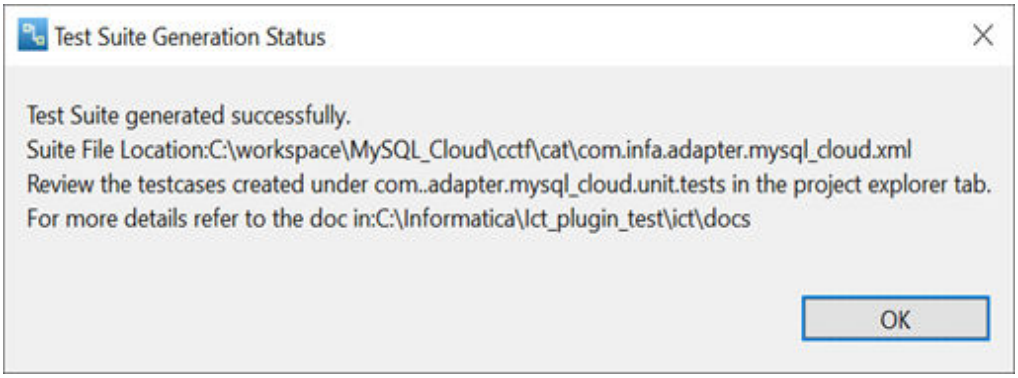
2. Enter the connector package version in the required field.



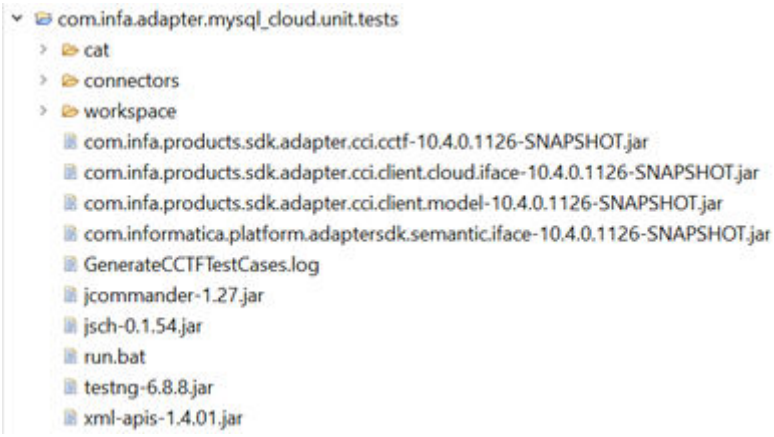
Ensure that you give the same version number that you specified in Phase 4 before you published the connector.

3. Click **Finish**

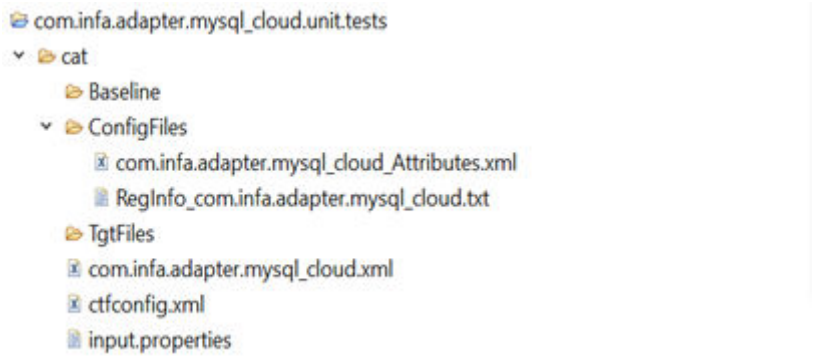
   A **Test Suite Generation Status** window appears.



4. Verify if the test suite is generated successfully and then click **OK**.
5. On the **Project Explorer** tab, expand the `com.infa.adapter.<connector_name>.unit.tests` folder.



6. Expand the **CAT** folder.



The following table describes the generated files by the test suite:

Files	Description
Attribute File	Generated as `com.infa.adapter.<adapter_name>_Attributes.xml` under the **ConfigFiles** folder. It consists of capability attributes such as read, write, and connection attributes. Specify the values for these properties.
TestNG Suite File	Generated as `com.infa.adapter.<adapter_name>.xml` under **cat folder**. It consists of test cases generated based on the connector capabilities. The test case parameter values are parameterized. You must specify the actual values for these parameters in each of the test cases before the XML runs. You can perform one of the following actions: - Edit the `testsuite` file to add the actual values for the parameters. - Specify the values in the `input.properties` file.

Files	Description
`ctfconfig.xml`	File that contains the required entries to run the test cases.
`input.properties`	Key-value pairs of parameterized entries in the testNG suite file. You must fill the actual values only for the required entries in the file. Do not remove any of the entries in this file.

Configuring the Parameters in the Test Suite File

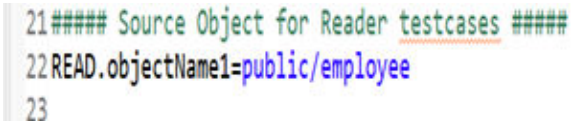
The test suite file includes all the test cases. To generate the test case with the required values, you must edit the `testsuite` file to replace the placeholder values with the required values, or you can specify the required values in the `input.properties` file.

You can perform one of the following tasks:

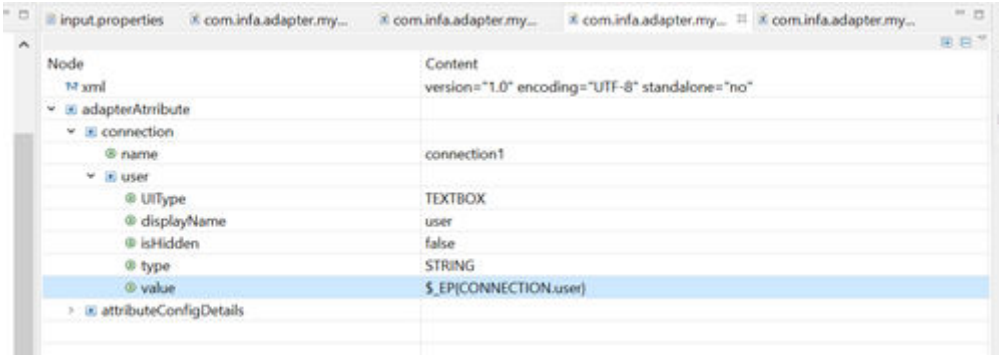
- Update the parameterized value for the object name in the test suite file.

- You can give the value of the object name through the `input.properties` file.
  The format of the value depends on the table name. If you choose the schema/folder and then select the object, specify the values in the following: `<schema/folder>/tablename`.

  See the following example where `public` is the schema and `employee` is the table name:



The `attribute.xml` file comprises all the capability parameter properties such as connection, read, write, and call. You can add new attributes or edit existing attributes using the `attribute.xml` files.



Regenerate a Test Case

You can regenerate a test case when new features are introduced in the connector.

If there are new features, regenerate to include the related new test cases in the existing test suite. When you regenerate, the previous files in the test suite are backed up and new test cases are created within the test suite.

When you select Phase 5 again on the **connector progress** tab, a new folder named **AdditionalTests** is created in the file `com.infa.adapter.<adapter_name>.unit.tests` of the project explorer.

For instance, if the connector does not support the filter property in the first version, the test suite will not have test cases related to the filter. However, if you add filter support in the next connector version, the filter test cases are generated in the **AdditionalTests** folder. You can review the test cases and add them to the existing suite.

Additionally, you will see a new testNG suite file, an attribute file, and an `input.properties` file with missing entries created under the **AdditionalTests** folder. Do not regenerate the test suite if there are no features added or supported by the connector.

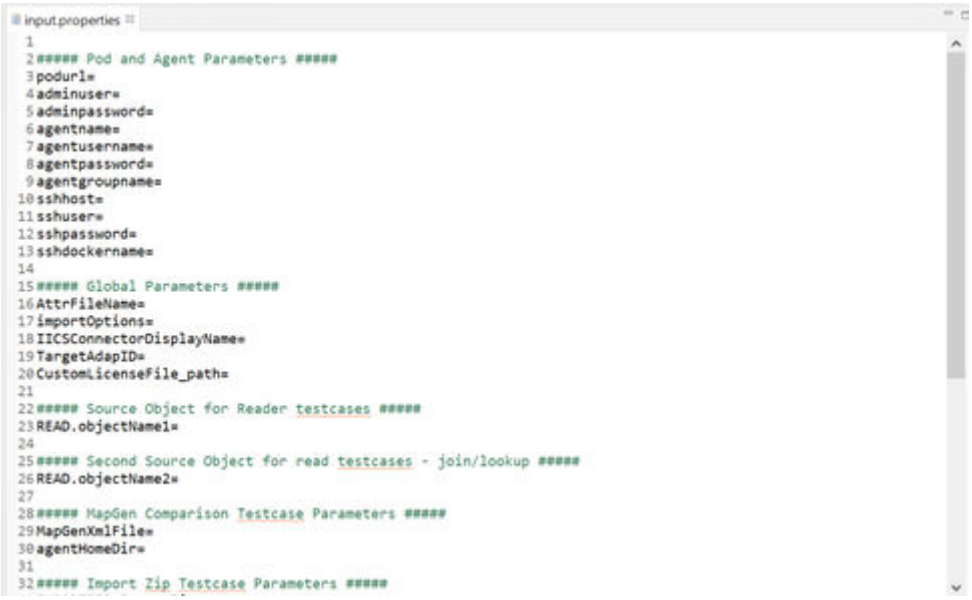
Running the Test Cases

After you have analyzed the connector package and generated the test cases, you can run the unit and integration tests.

You can then view the success or failure status in the test report.

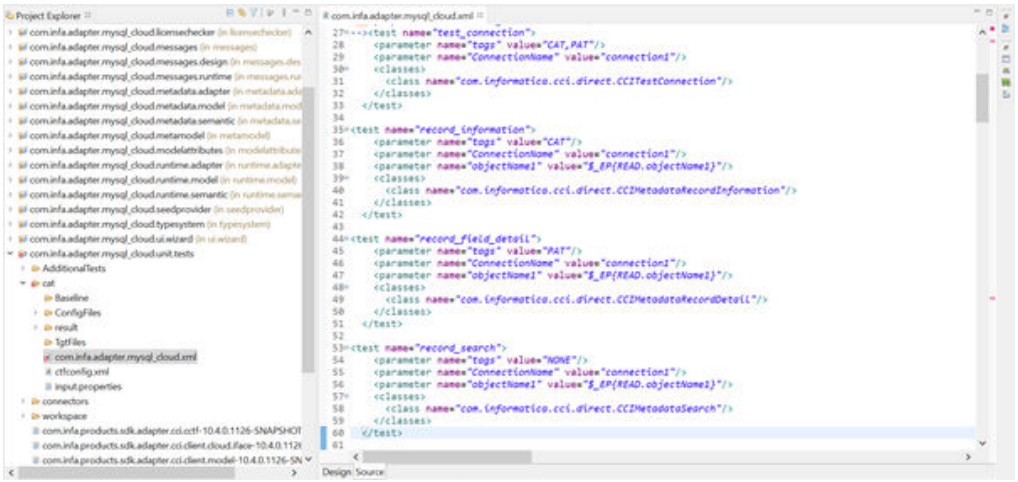
Running the Unit Test

Enter the values in the `input.properties` file present in the CAT folder before you run the unit test. The file contains a list of key-value pairs of parameterized entries in the `testng` suite file. You can fill the actual values for only the required entries in this file.



Perform the following steps to run the unit test:

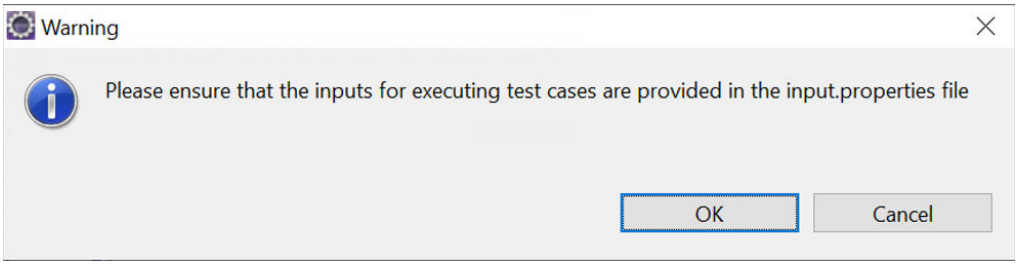
1. Edit the tag values in the following suite: CAT for Code Acceptance Test or Unit Test and PAT for Product Acceptance Test or Integration Test.



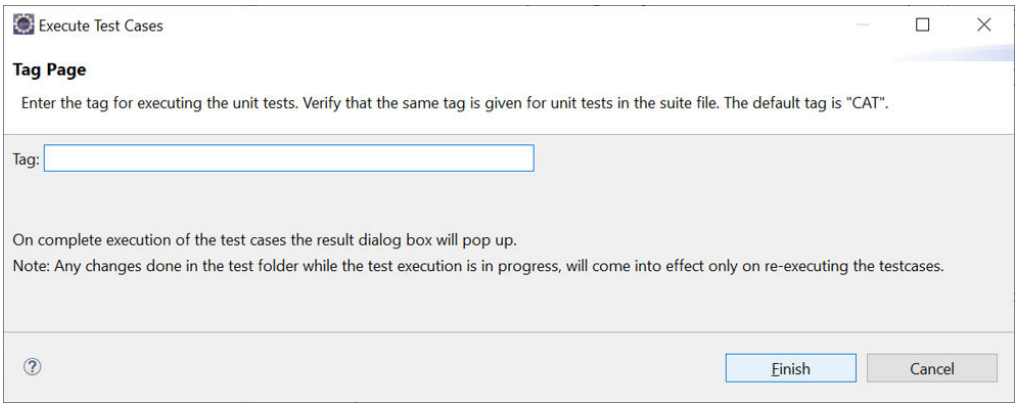
   Set the value of tags to **NONE** during test case review to skip the test case execution.

2. Switch to the **ICT perspective** window.
3. Click **Execute Unit Test** in Phase 5 on the **Connector Progress View** tab.

   A warning window appears.

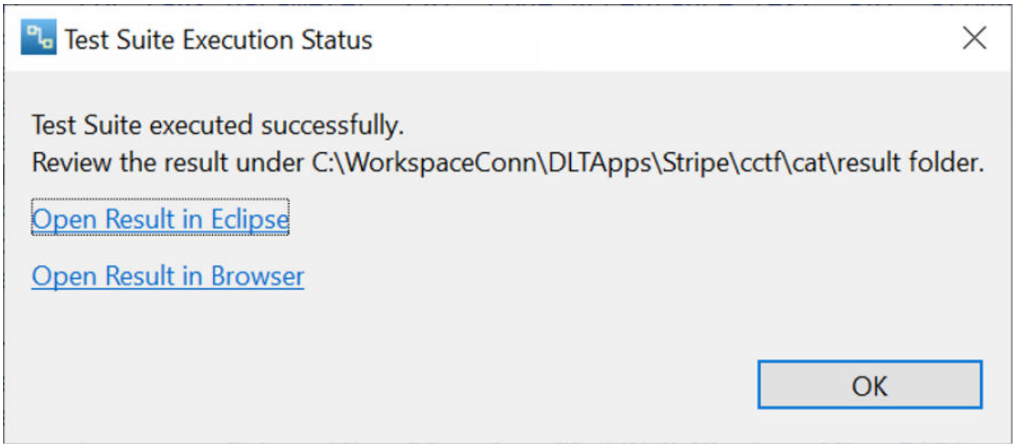


4. Click **OK**.
5. In the Execute Test Cases window, specify the tags to execute the unit test cases.

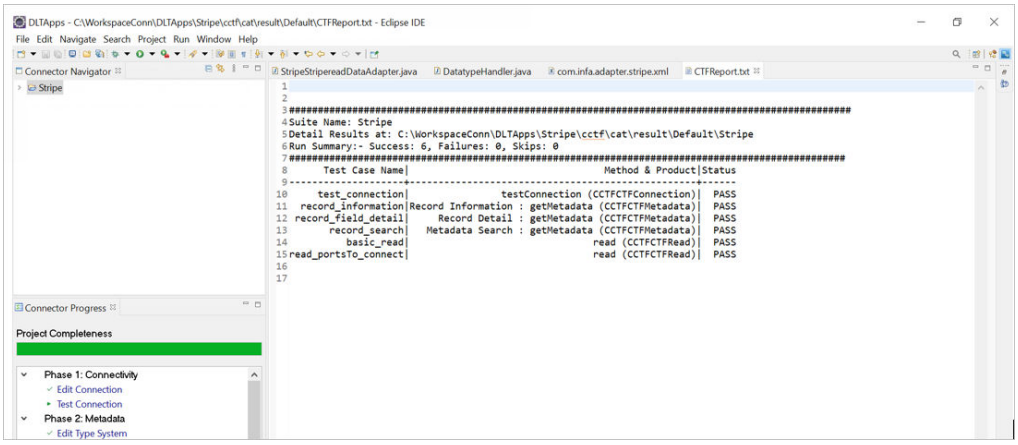


6. Click **Finish**.

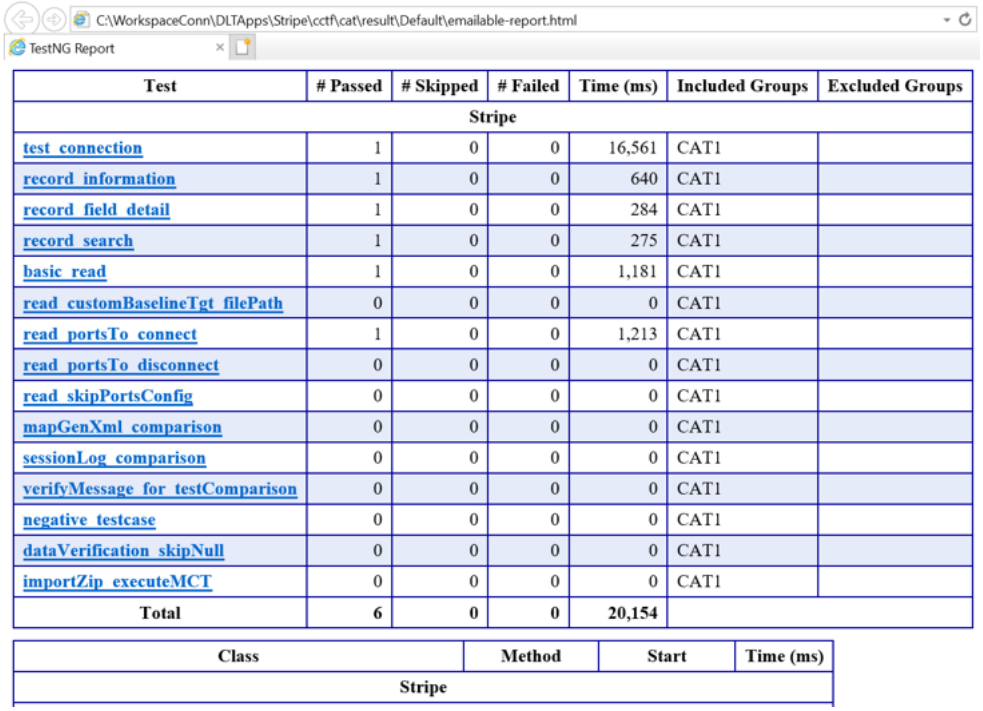
   When the unit test run is complete, the following status window appears:



7. Click **Open Result in Eclipse** to view the results in Eclipse.



Optionally, click **Open Result in Browser** to view the result in the browser.



Test	# Passed	# Skipped	# Failed	Time (ms)	Included Groups	Excluded Groups
Stripe						
test_connection	1	0	0	16,561	CAT1	
record_information	1	0	0	640	CAT1	
record_field_detail	1	0	0	284	CAT1	
record_search	1	0	0	275	CAT1	
basic_read	1	0	0	1,181	CAT1	
read_customBaselineTgt_filePath	0	0	0	0	CAT1	
read_portsTo_connect	1	0	0	1,213	CAT1	
read_portsTo_disconnect	0	0	0	0	CAT1	
read_skipPortsConfig	0	0	0	0	CAT1	
mapGenXml_comparison	0	0	0	0	CAT1	
sessionLog_comparison	0	0	0	0	CAT1	
verifyMessage_for_testComparison	0	0	0	0	CAT1	
negative_testcase	0	0	0	0	CAT1	
dataVerification_skipNull	0	0	0	0	CAT1	
importZip_executeMCT	0	0	0	0	CAT1	
Total	**6**	**0**	**0**	**20,154**		

Class	Method	Start	Time (ms)
Stripe			

8. To view the detailed log, see the `ctfrun.log` file in the result folder.

   The folder also contains the `testng` and `junit` reports.

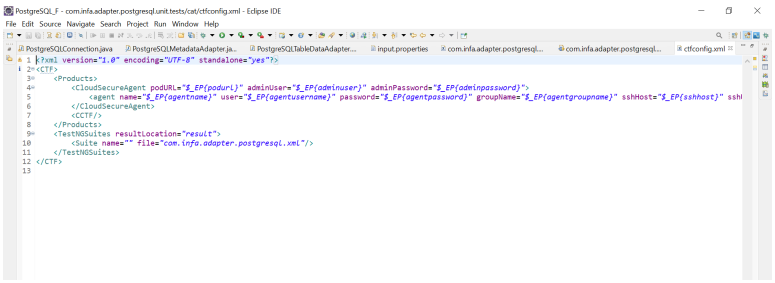
Running the Integration Test

Perform the following steps to run the integration test:

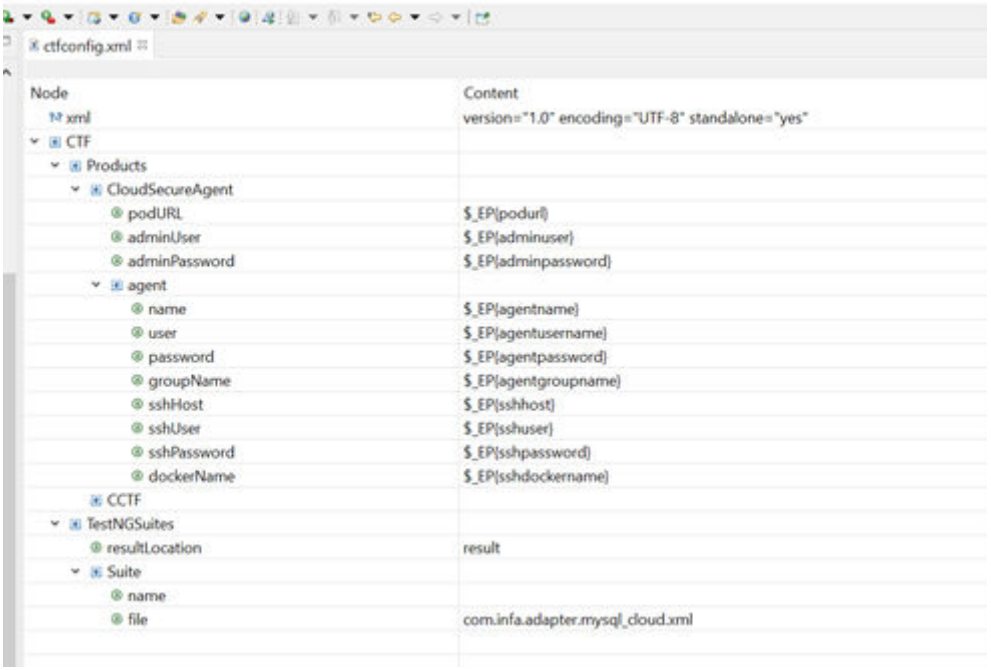
1. On the **Project Explorer** tab, open the `ctfconfig.xml` from the **CAT** folder.

The `ctfconfig.xml` file consists of the following configuration details:

- The product details such as Agent name, Pod URL, Admin user, and Admin Password.
- Test suites for running the tests. You can list more than one test suite.



2. You must enter the pod details in the `ctfconfig.xml` file or in the `input.properties` file.



The following table describes the keys in the `ctfconfig.xml` file:

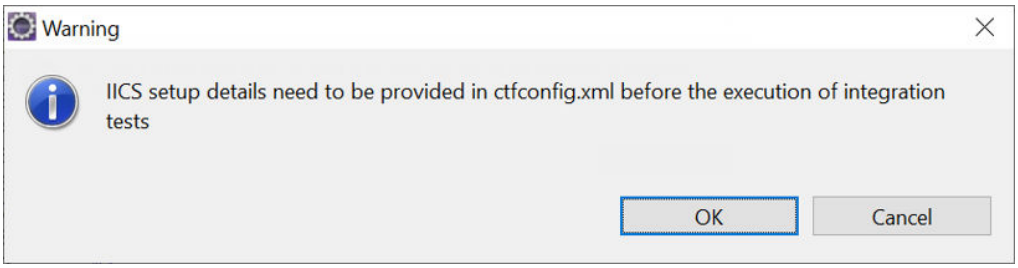
Key	Description
podurl	Pod URL for logging into the pod.
adminuser	Admin user for the pod. This parameter is required to assign the required licenses to execute the test case. You can leave the value blank if the required licenses are already assigned to the organization user.

Key	Description
adminpassword	Admin password.
agentname	Agent name that appears on the pod.
agentusername	Agent user name.
agentpassword	Agent password.
agentgroupname	Agent group name that appears on the pod.
sshhost	Host IP address of the agent machine.
sshuser	Host user name of the agent machine.
sshpassword	Host password of the agent machine.
sshdockername	Docker name. Specify the docker name if the agent is installed on the docker machine.

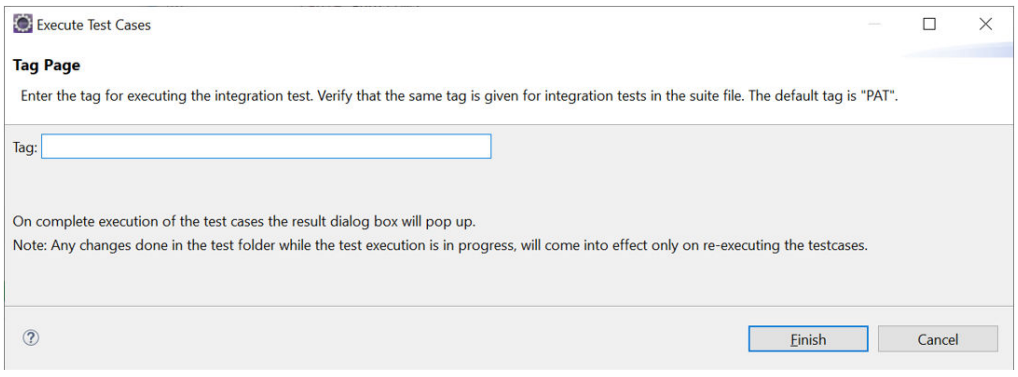
3. Edit the tag values in the testNG suite: CAT for Code Acceptance Test or Unit Test and PAT for Product Acceptance Test or Integration Test.

   Set the value of tags to NONE during test case review to skip the test case execution.
4. Switch to the **ICT perspective** window.
5. Click **Execute Integration Test** in Phase 5 on the **Connector Progress View** tab.

   A warning window appears.

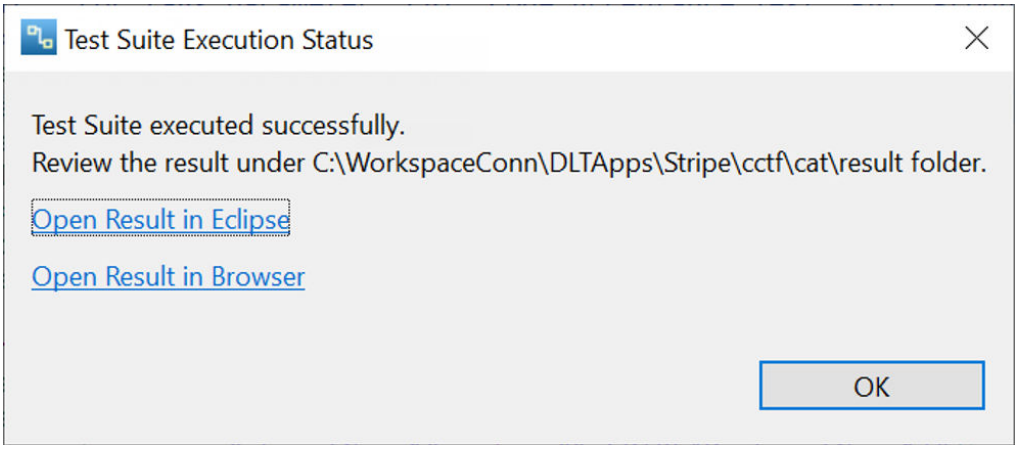


6. Click **OK**
7. In the Execute Test Cases window, specify the tags to execute the integration test cases.



8. Click **Finish**.

   When the integration test run is complete, the following status window appears:



9. Click **Open Result in Eclipse** to view the results in Eclipse.

   Optionally, click **Open Result in Browser** to view the results in the browser.

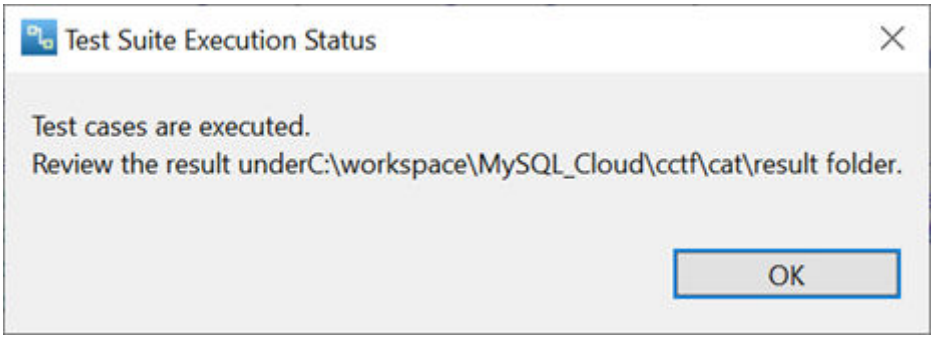
10. To view the detailed log, see the `ctfrun.log` file in the result folder.

    The folder also contains the `testng` and `junit` reports.

Failure Scenario

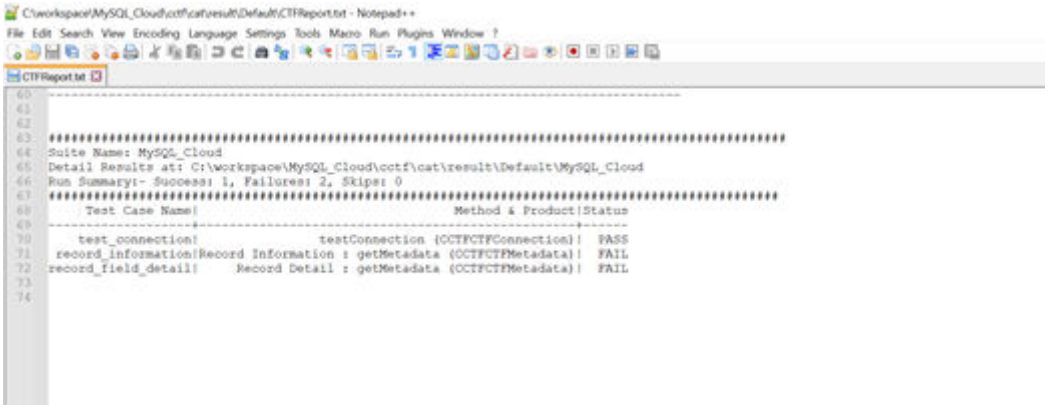
After you execute the test cases in unit and integration tests, you can check the reports to view the success or failure status of the tasks.

After you execute the test, a status window appears. If the execution is complete, the following window appears.



Go to the file path in the status window and navigate to the result folder. In the folder, open the `CTFReport.txt` file to view the result of executing the tests. PASS or FAIL status is displayed for the corresponding test cases.

The following image displays the PASS and FAIL status for the test cases.


C:\workspace\MySQL_Cloud\cctf\cat\result\Default\CTFReport.txt - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CTFReport.txt
Suite Name: MySQL_Cloud
Detail Results at: C:\workspace\MySQL_Cloud\cctf\cat\result\Default\MySQL_Cloud
Run Summary:- Success: 1, Failures: 2, Skips: 0
Test Case Name| Method & Product|Status
test_connection| testConnection (CCTFCTFConnection)| PASS
record_information|Record Information : getMetadata (CCTFCTFMetadata)| FAIL
record_field_detail| Record Detail : getMetadata (CCTFCTFMetadata)| FAIL

CHAPTER 13

Connector Example: MySQL_Cloud

This chapter includes the following topics:

MySQL_Cloud Connector Overview

The Informatica Connector Toolkit includes sample source code of the MySQL_Cloud Connector. You can use the MySQL_Cloud Connector sample source code as a model to develop a connector for a data source.

Note: The sample MySQL_Cloud Connector is for illustration purposes only.

MySQL_Cloud Connector Requirements

To run the sample connector, you must install the MySQL JDBC driver on your development machine. Use the MySQL JDBC driver to access the metadata and perform read and write operations on the MySQL database. Download MySQL JDBC driver version 8.0.13 or later from the following URL:

```
http://dev.mysql.com/downloads/connector/j
```

Building the Sample Connector

You can use the sample connector source code to develop a connector for a data source. Import the sample connector project in to the Eclipse IDE and use the Informatica Connector Toolkit to publish or deploy the sample connector.

To use the sample source code and develop a connector, perform the following tasks:

1. From the Eclipse IDE, click **File** > **Import**. The **Import** dialog box appears.
2. Select **Existing Projects into Workspace** and then click **Next**. The **Import Projects** page appears.
3. Select **Select root directory** and browse to the directory of the sample connector that you want to import into Eclipse.
4. Click **Finish**.
   The sample connector project appears in the package explorer.
5. Change to the **Informatica Connector** perspective.
6. Edit the connection, types system, metadata, and run-time components, if required.
7. Publish the sample connector or deploy the sample connector on the Cloud Data Integration service.

For more information, see the *Sample Adapter Readme*.

MySQL_Cloud Connector Components

The sample MySQL_Cloud Connector includes the following components:

Contribution

Use the contribution plug-in project to get information on plug-ins that contribute to the MySQL_Cloud Connector project. The name of the MySQL_Cloud contribution plug-in is `com.infa.adapter.mysql_cloud.adapter.contribution`.

Connection model

Use the connection model Java project to represent the connection model for the MySQL_Cloud Connector. The name of the MySQL_Cloud connection model Java project is `com.infa.products.adapters.mysql_cloud.models.connection.annotatedjava`.

Connection adapter

Use the connection adapter plug-in project to provide connection attribute information and consumer information for the MySQL_Cloud Connector. The name of the MySQL_Cloud connection attributes plug-in is `com.infa.products.adapter.mysql_cloud.connection.adapter`.

Seed provider

Use the seed provider Java project to map native data types to Informatica data types. The name of the MySQL_Cloud seed provider Java project is `com.infa.adapter.mysql_cloud.seedprovider`.

Type system

Use the type system plug-in project to contribute the seed provider of the connector to the Informatica Intelligent Cloud Services platform. The name of the MySQL_Cloud type system plug-in project is `com.infa.products.adapter.mysql_cloud.typesystem`.

Metadata model

Use the metadata model Java project to represent the metadata model for the MySQL_Cloud Connector. The name of the MySQL_Cloud metadata model Java project is `com.infa.products.adapters.mysql_cloud.models.metadata.annotatedjava`.

Metadata adapter

Use the metadata adapter Java project to provide the functionality to open and close connections to the MySQL_Cloud Connector. The name of the MySQL_Cloud metadata adapter Java project is `com.infa.products.adapter.mysql_cloud.metadata.adapter`.

Run-time model

Use the run-time model Java project to represent the run-time model for the MySQL_Cloud Connector. The name of the MySQL_Cloud run-time model java project is `com.infa.products.adapters.mysql_cloud.models.runtime.annotatedjava`.

Run-time adapter

Use the run-time adapter plug-in project to implement run-time adapter for the MySQL_Cloud Connector in Java. The name of the MySQL_Cloud run-time adapter plug-in project is `com.infa.products.adapter.mysql_cloud.runtime.adapter`.

Run-time pdo

Use the run-time pdo plug-in project to implement run-time pushdown optimization for the MySQL_Cloud Connector in Java. The name of the MySQL_Cloud run-time pdo plug-in project is `com.infa.products.adapter.mysql_cloud.runtime.pdo`.

Metamodel bundle

Use the MySQL_Cloud metamodel bundle plug-in project to list the metaclasses for connector packages such as connection, metadata, run-time ASO, and run-time capability. The name of the MySQL_Cloud metamodel bundle plug-in project is `com.infa.products.adapter.mysql_cloud.metamodel`.

Model attributes

Use the model attributes plug-in project to define the presentation labels for the field, record, and run-time extensions. The name of the MySQL_Cloud model attributes plug-in project is `com.infa.products.adapter.mysql_cloud.modelAttributes`.

Design-time messages

Use the messages design plug-in project to implement design-time messages for the MySQL_Cloud Connector in Java. The name of the MySQL_Cloud messages design plug-in project is `com.infa.products.adapter.mysql_cloud.messages.design`.

Run-time messages

Use the messages runtime plug-in project to implement run-time messages for the MySQL_Cloud Connector in Java. The name of the MySQL_Cloud messages runtime plug-in project is `com.infa.products.adapter.mysql_cloud.messages.runtime`.

Library information

Use the library info plug-in project to define the run-time adapter based on the programming language in which you implement the run-time adapter. Currently, you can use only Java interfaces to implement the run-time connector. The name of the MySQL_Cloud library Info plug-in project is `com.infa.products.adapter.mysql_cloud.libraryInfo`.

License

Use the license Java project to perform license checks for the MySQL_Cloud Connector. The name of the MySQL_Cloud license Java project is `com.infa.products.adapter.mysql_cloud.license`.

UI wizard

Use the UI wizard project to define the icons for the Import wizard. The UI wizard project is a Java project. Cloud Data Integration uses the import options to display UI components when a connector consumer creates a data object. The name of the UI wizard project is `com.infa.products.adapter.mysql_cloud.wizard`.

Feature

Use the feature plug-in project, which is an Eclipse feature, to define a connector plug-in. The name of the MySQL_Cloud feature plug-in project is `com.infa.products.adapter.mysql_cloud.feature`.

CHAPTER 14

Version Control Integration

This chapter includes the following topics:

Git Version Control Integration

You can integrate Git version control to store new or existing connector projects in a Git repository.

You can use version control to effectively monitor and track the changes in the connector bundle over different release cycles. After you enable, you can recall version specific files and deploy them into your current projects.
You can configure the connector project in a local or a remote Git repository. After you integrate Git with the Eclipse IDE, you can commit new connector bundles or existing connector bundles to the Git repository.

Prerequisites

Before you integrate Git with Eclipse IDE, complete the following prerequisites:

- Install Git version 2.33.
- Set the custom property `UseGit=true` in the `env.properties` file in the following directory: `<eclipse_home>/dropins/com.informatica.tools.ui.ict`
- If Eclipse is already running, restart Eclipse.

Build a Connector in Git Repository

You can build a connector in the Git repository and recall version specific files and deploy them into your current projects.

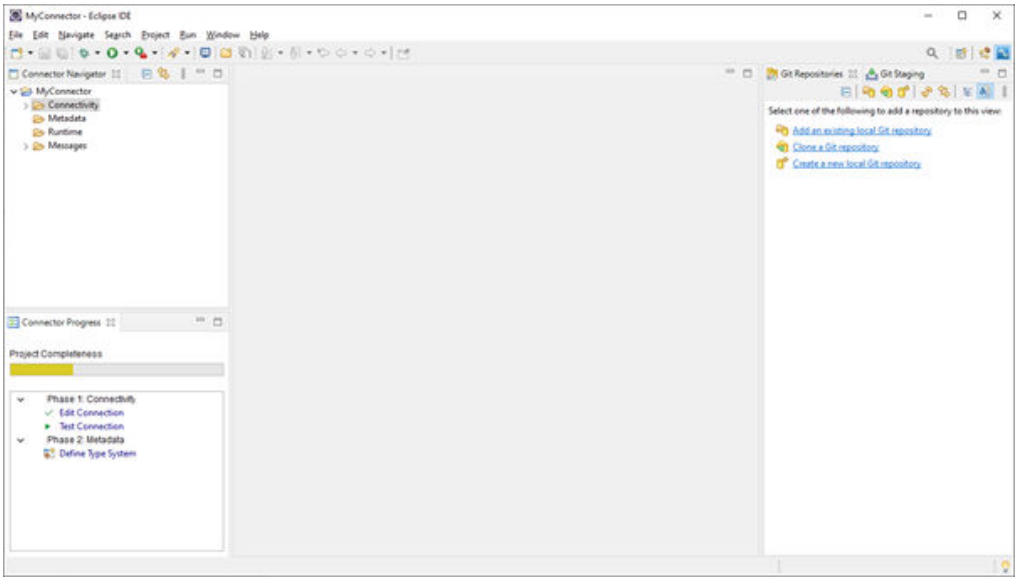
1. In the Eclipse IDE, create a new connector project.
2. Click **File > New > Project**.

   A **New Project** window appears.
3. Enter the project name in the **Project Name** field.
4. Click **Finish**.
5. Select **Window> Perspective> Open Perspective> Other**.

   A window appears with a list of open perspectives.

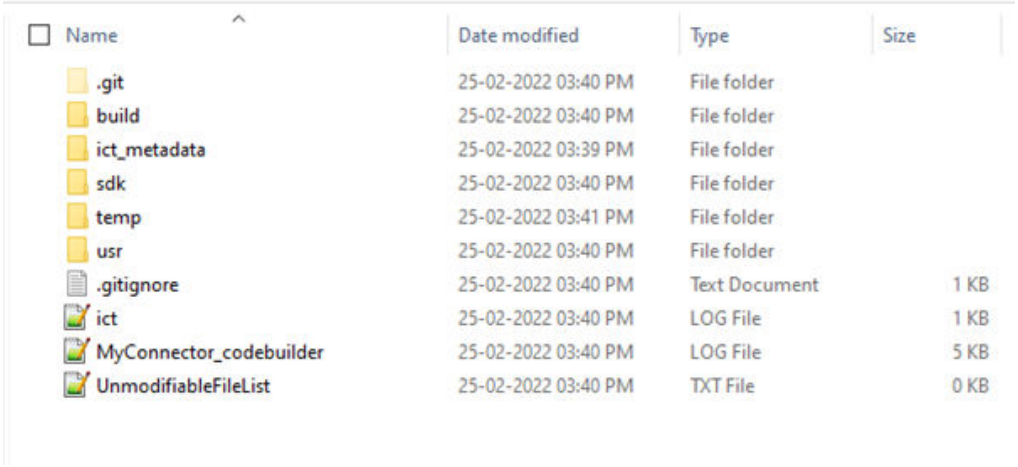
6. Select **Informatica Connector** from the list of perspectives.

   The Git repositories and Git staging tabs are displayed in the Informatica connector perspective:

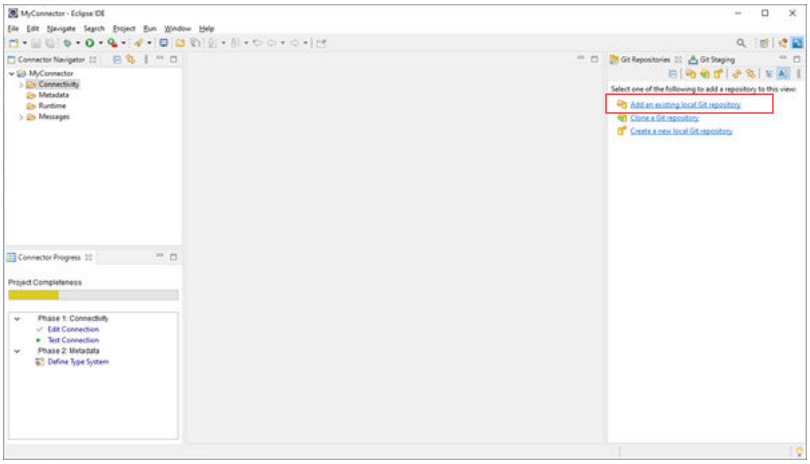


7. Follow the instructions from Chapter 3 to create a new connector.

   After you create the connector the project workspace is initialized as a `.git` file in the connector workspace:

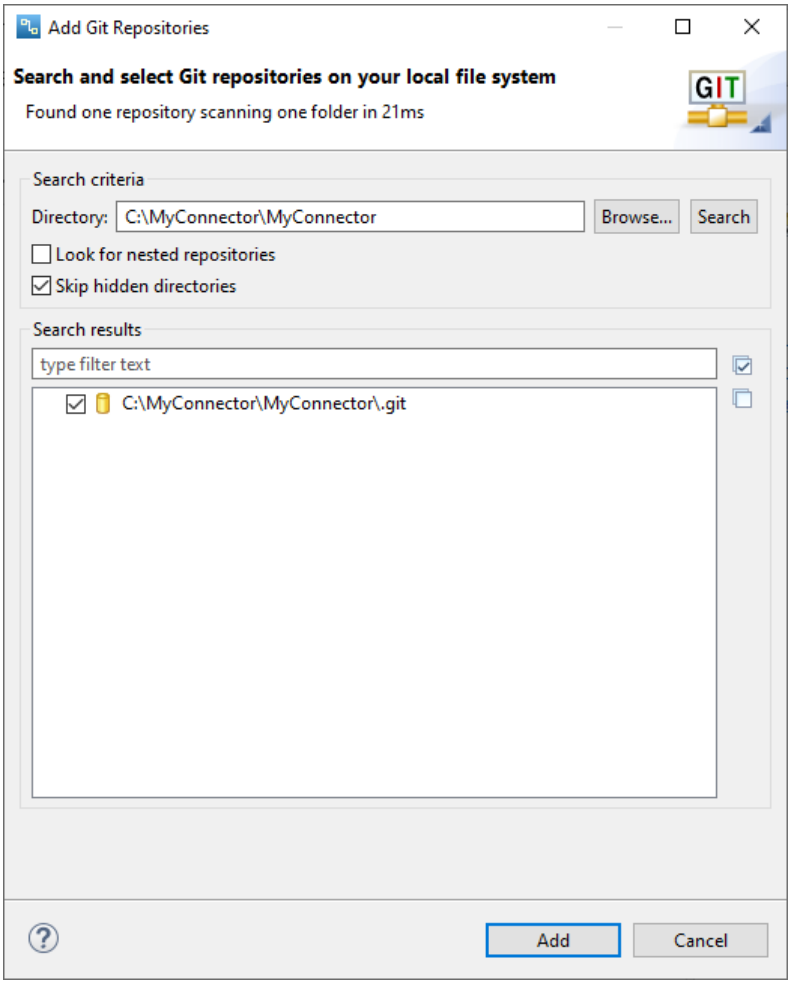


8. Select the **Add an existing local Git repository** option on the **Git Repositories** tab.



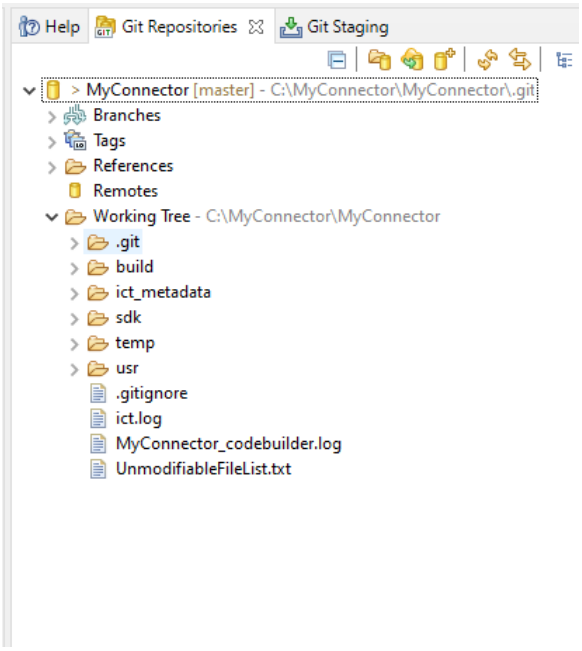
A search window appears.

9. Select the required connector workspace from your local system.



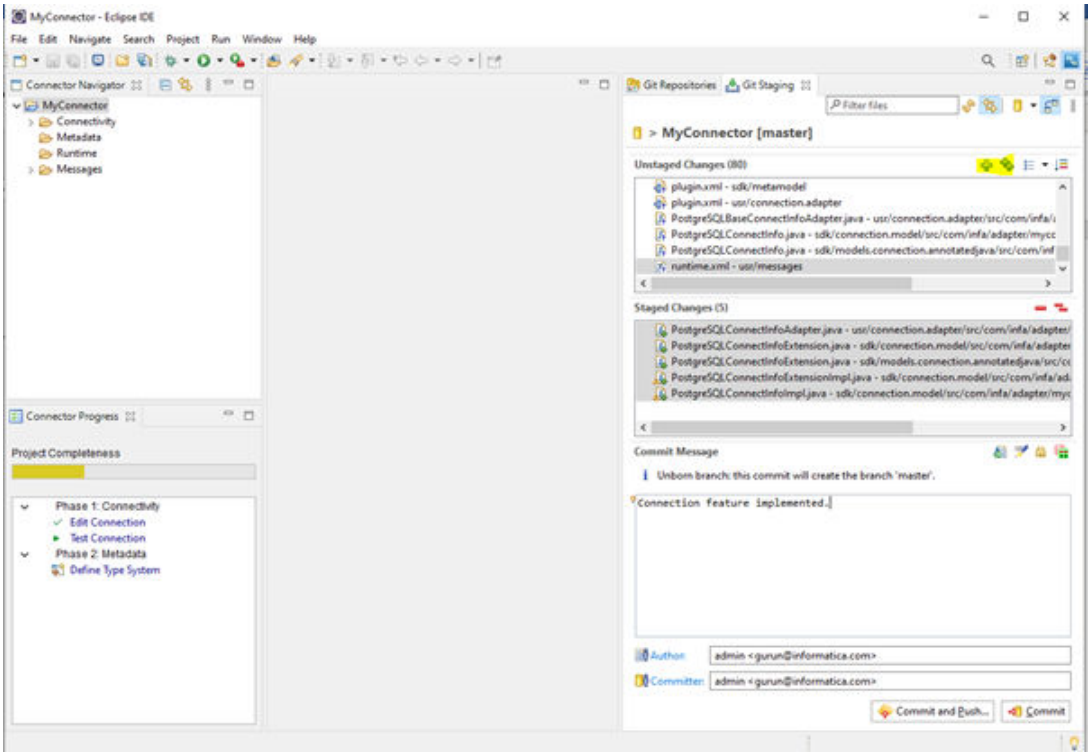
10. Select the file and click **Add**.

11. Expand the **Working Tree** to view the files in the connector bundle.



The **Git Staging** tab displays staged and unstaged files. The selected files are listed under **Staged Changes**.

12. Add a commit message and click **Commit.**



The connector workspace is saved in a local Git repository.

13. To provide the remote Git repository details to save your connector workspace in a remote Git repository, click **Commit and Push**.

Perforce Version Control Integration

You can integrate Perforce version control to store new or existing connector projects in a Perforce server.

You can use version control to effectively monitor and track the changes in the connector bundle over different release cycles. After you enable, you can recall version specific files and deploy them into your current projects.
Download and install Perforce Eclipse plugin on the Eclipse IDE. Then, integrate the connector source code with the Perforce server.

Prerequisites

Before you integrate Perforce with Eclipse IDE, complete the following prerequisites:

- Download and install the Perforce Eclipse plugin.
- Set the environment variable `P4IGNORE=p4ignore.txt`.
- Set the custom property `UsePerforce=true` in the `env.properties` file in the following directory: `<eclipse home>/dropins/com.informatica.tools.ui.ict`

Download and Install the Perforce Eclipse Plugin

Install the Perforce Eclipse plugin on the Eclipse IDE. You can install the Perforce Eclipse plugin from the Perforce location or from a .zip file.

1. In Eclipse, navigate to **Help** > **Install New Software**.

   The **Install** window appears.
2. Click **Add**.

   The **Add Repository** window appears.
3. Enter the name and location of the repository.

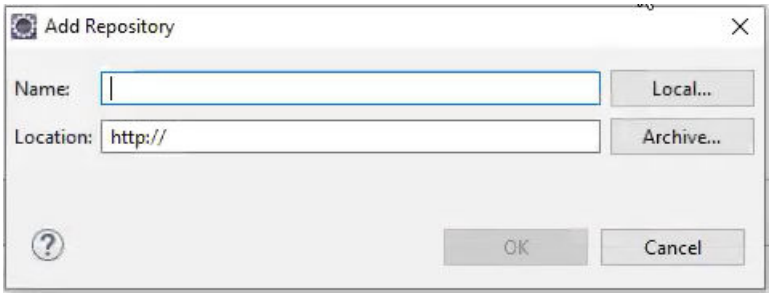
   For example, enter the following location to install the Perforce Eclipse plugin for Eclipse 4.7:

   http://www.perforce.com/downloads/http/p4-eclipse/install/4.7

   You can also click **Archive** and select the Perforce Eclipse plugin zip file from the following location:

   https://www.perforce.com/downloads/helix-plugin-eclipse-p4eclipse

   The following image shows the how to add a repository:



4. Click **OK**.

   The **Install** window appears.
5. Select the **Perforce Team Provider (Core)** plugin, and click **Next**.

   The **Feature License** page appears.

6. Select the licenses and click **Finish** to install the plugin.
7. Restart Eclipse.

Build a Connector in Perforce

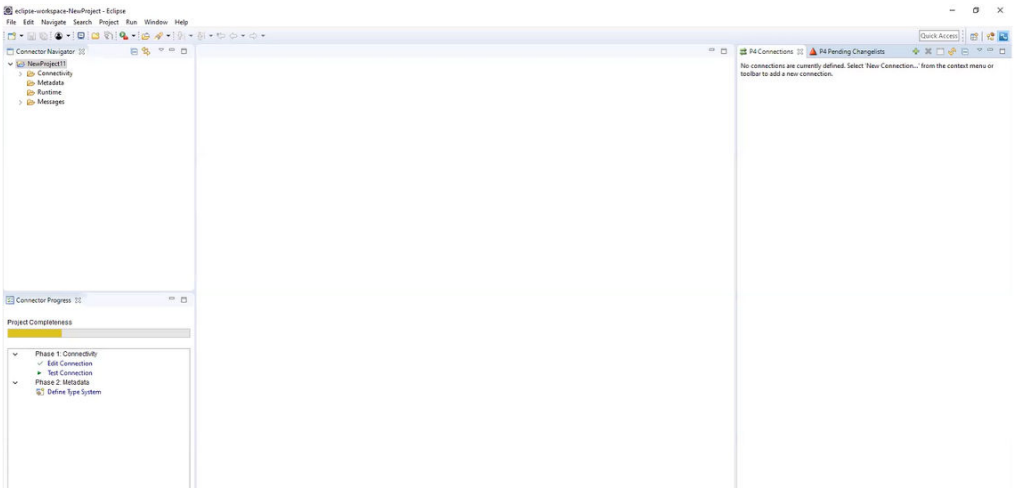
In the Eclipse IDE, create a new connector project.

1. Click **File** > **New** > **Project**.

   The **New Informatica Connector Project** page appears.
2. Enter the connector ID, connector name, vendor ID, and vendor name.
3. Click **Finish**.
4. Select **Window** > **Perspective** > **Open Perspective** > **Other**.

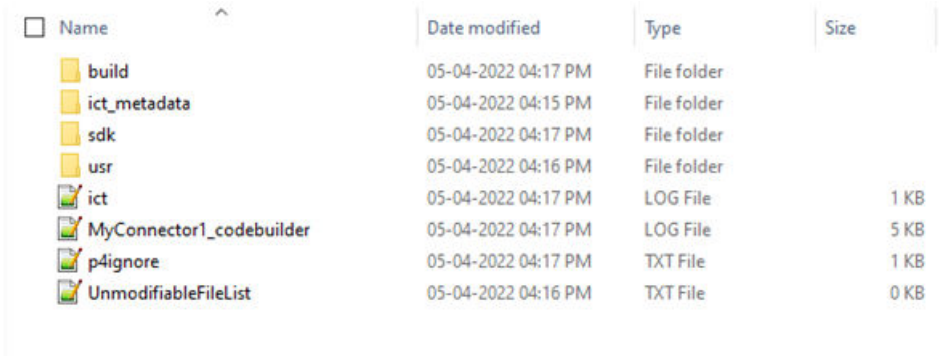
   A window appears with a list of open perspectives.
5. Select **Informatica Connector** from the list of perspectives.

   The P4 connections view and P4 pending changelists view are displayed in the Informatica Connector perspective:



6. Perform the steps from Chapter 3 to create a new connector.

   After you create the connector, you will see a `p4ignore` file in the connector workspace:

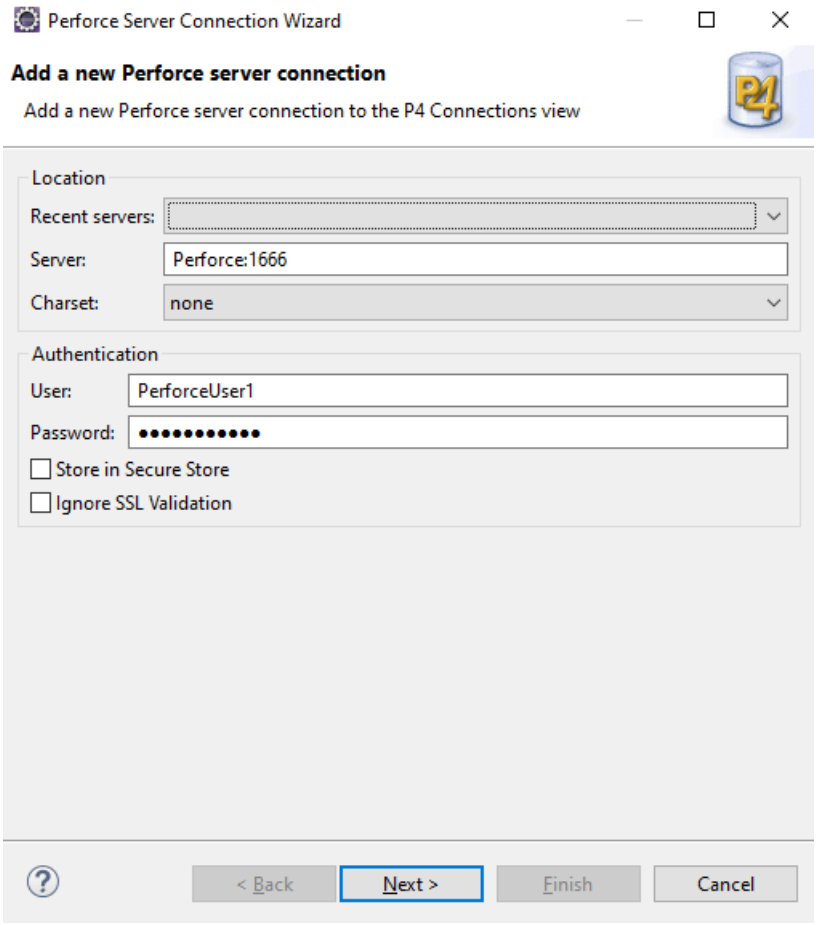


   Use the `p4ignore` file to exclude the files such as the generated class files, log files, and temp files from being tracked for checkin.
7. Right click on the P4 Connections view and click **New connection**.

The **Perforce Server Connection Wizard** window appears.

8. Specify the server details and the perforce user account details.

   The following image shows the perforce server connection page:



9. Click **Next**.
10. Select an existing Perforce workspace or create a new Perforce workspace.

    To create a new Perforce workspace, enter the workspace name. Enter the location or browse to the eclipse workspace of the current project and select the folder.
11. Clear the **Launch the Perforce Project Import Wizard** check box, and click **Finish**.

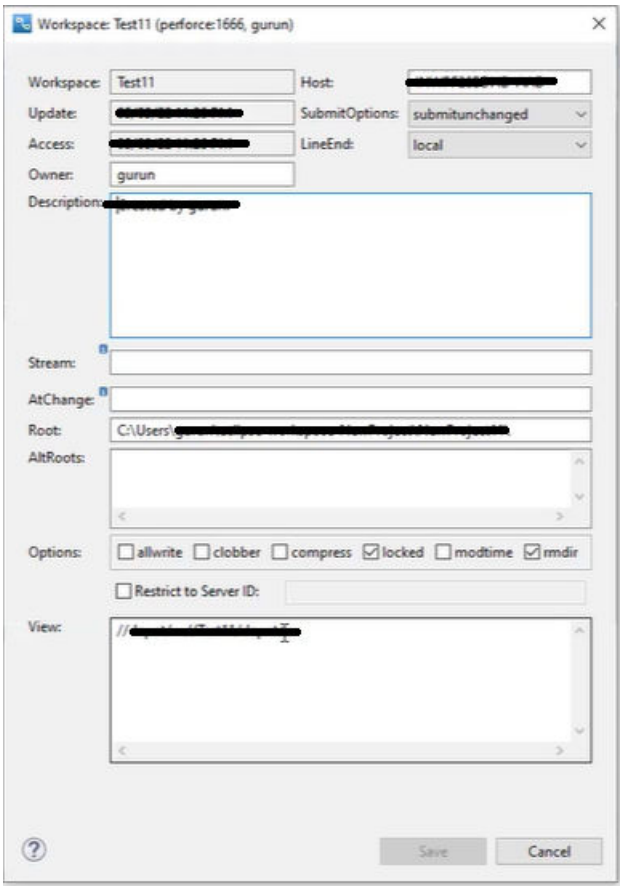
    The connection is created.
12. In the P4 Connections view, right click on the connection, and select **Edit Perforce Workspace**.

    The **Workspace** window appears.
13. In the **View** tab, map the Perforce workspace that you created to the Perforce depot.

    Use the following format to map the Perforce workspace:

    ```
    //depot/MyProjects/... //MyConnector1/...
    ```

    The following image shows where you can map the Perforce workspace:



14. Click **Save**.

    The **Perforce Workspace** window appears.

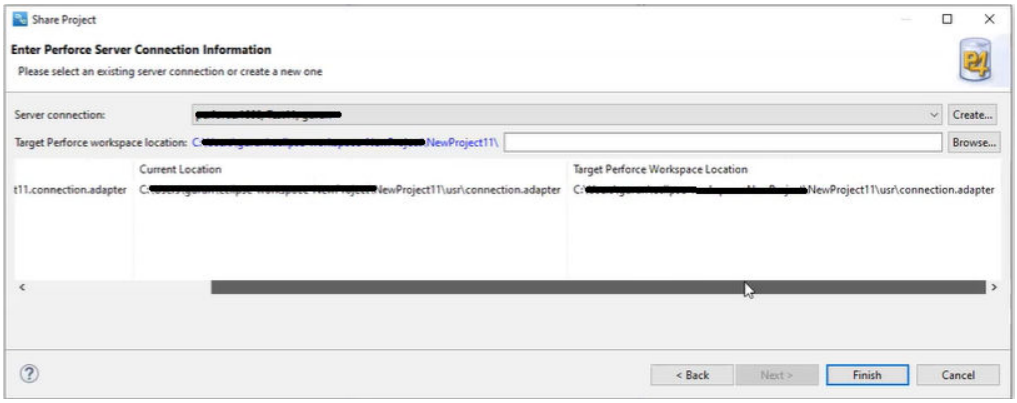
15. Select the **Do not synchronize the projects** check box, and click **OK**.
16. Select **Window** > **Perspective** > **Open Perspective** > **Java**.
17. In the **Project Explorer** pane, select all the Java projects, right click, and navigate to **Team** > **Share Project**.

    The **Share Project** window appears.

18. Select the Perforce repository type, and click **Next**.
19. Select the Perforce connection from the drop down.

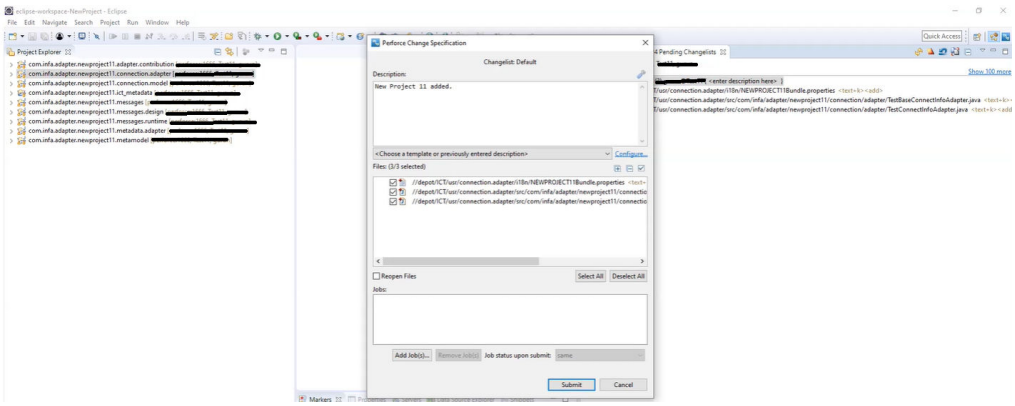
    Ensure that the current location and the target Perforce workspace location point to the same folder.

    The following image shows the Perforce connection and the target Perforce location:



20. Click **Finish**.
21. In the **Project Explorer** pane, select all the Java projects, right click, and navigate to **Team** > **Mark for Add**.
22. Go to the P4 pending changelists view and you can see all the Java source code and resource files are listed in the default change list.
23. Right click on the default change list, and click **Submit**.
24. Specify the change list description, and click **Submit**.

    The following image shows how to submit the files from the P4 pending changelist:



    The connector source code is integrated with the Perforce server.
25. In the **Project Explorer** pane, you can right click on the java or resource file and make further changes to the files, such as check out, mark for delete, and get latest revision.

    You can switch back to the ICT view and continue with the connector development.

APPENDIX A

Metadata Models

This appendix includes the following topics:

Metadata Model Overview

The Informatica Connector Toolkit internally uses metadata models to define native metadata objects for the connector to read from and write to the data source. The metadata model contains components that represent the metadata of the data source.

The Informatica Connector Toolkit uses pattern classes in the metadata model to describe the metadata structure of the data source. The metadata model uses a pattern template to define the metadata pattern of a connector.

Note: Do not use the metadata models directly to define the native metadata objects for the connector. Use the Informatica Connector Toolkit to define native metadata objects.

Metadata Model Components

The metadata model contains components that represent the metadata of the data source.

The metadata model consists of the following components:

Flat Record

A flat record represents a structure that contains columns, unique keys, and primary keys. The structure of a flat record is similar to a database table that contains columns and keys.

A flat record contains attributes that store the following information:

- Name of the native metadata object
- Type of native metadata object
- Related records for the flat record

- Primary key for the flat record
- Unique keys for the flat record
- Indexes for the flat record
- Any additional record attributes specific to the data source

Field

A field is a data structure for a single unit of data in a data source.

A field contains attributes that store the following information:

- Name of the field
- Default value of the field
- Precision of the field
- Scale of the field
- Boolean value that indicates whether the field can contain a null value
- Any additional field attributes specific to the data source

Constraints

Constraints represent the primary key and unique keys for a flat record.

The primary key and unique key contain attributes that store the following information:

- Name of the key
- Native name of the key defined in the native metadata
- List of fields that form the key

Index

Index represents a native index that orders the flat records or uniquely identifies a row in the flat record.

An index contains attributes that store the following information:

- Name of the index
- Native name of the index
- Boolean value that indicates whether the index is unique
- List of index fields
- Index order to retrieve the data

The Informatica Connector Toolkit internally uses the metadata model components to represent the data source metadata and persists the metadata in the Model repository.

Metadata Patterns

The metadata pattern of a connector describes the metadata structure of the native data source. You can choose a metadata pattern template to define the metadata pattern of a connector.

A metadata pattern template is a set of packages such as catalog, container, flat record, and so on. Sets of related classes that define the metadata pattern are organized as packages.

Though a connector can have more than one metadata pattern, the Informatica Connector Toolkit currently supports only Type A template. The type A template provides the metadata patterns for flat records. You can

use the metadata catalog that the Type A template provides to describe the metadata structure of the data source.

Features of Type A Metadata Template

The Informatica Connector Toolkit uses the Type A metadata pattern template to define the metadata pattern for flat records.

The Type A metadata pattern template supports the following features:

Catalog

A catalog is a container for metadata objects that you can import.

Container

A container is a collection of packages.

Package

A package contains flat records. A package may also contain other packages.

Flat record

A flat record represents the structure of the native metadata object that you can import. The flat record structure contains fields, relationships, index, and constraints.

Field

A field is a data structure for a single piece of data in a data source.

Record relationship

A record relationship represents the relationship between flat records.

Index

An index represents a native index that orders the flat records or uniquely identifies a flat record.

Constraints

A constraint represents the unique key and primary key for a flat record.

APPENDIX B

ASO Model

This appendix includes the following topics:

ASO Model Overview

The Informatica Connector Toolkit internally uses an Adapter Specific Object (ASO) model to represent operations on native metadata objects.

The ASO model serves as a container for native metadata objects and associates operations such as read, write, or lookup with the native metadata objects. A native metadata object represents the native importable metadata of a data source such as a flat file, relational data source, and nonrelational data source. An ASO model contains references to multiple native metadata objects and operations.

Note: Do not use the ASO models directly to define operations on native metadata objects. Use the Informatica Connector Toolkit to define operations on native metadata objects.

ASO Model Components

Use the Informatica Connector Toolkit to define the components that comprise the application specific object.

The ASO model consists of the following components:

ASO

The ASO object represents the generic connector specific object. The ASO object contains references to the following components that comprise the ASO object.

- ASO Operation
- Catalog
- Projection

ASO Operation

The ASO operation object contains references to capabilities, capability attributes, and complex types that you associate with an operation.

Catalog

The catalog object represents the native metadata object. The native metadata object is the native importable metadata of a data source. The catalog object contains references to components of native metadata objects such as columns, unique keys, and primary keys.

Projection

The projection object represents the operations that you can perform when you read or write data. The projection object contains references to the list of operations, projection type, and basic projection view.

ASO Projections

An ASO projection is a sequence of operations that you perform on data when you read data from a data source or write data to a data source.

Use the Informatica Connector Toolkit to define a basic ASO projection model or an advanced projection model according to your requirements. The basic ASO projection model provides a simple view of the advanced projection model. The Informatica Connector Toolkit uses the BasicProjectionView interface to provide a simple and basic model to the connector developer. Use the get methods in the BasicProjectionView interface to get information on native metadata object, platform types, scale, precision, and conditions like filter and join.

To define the advanced ASO projection model, define the following projection operation interfaces as required.

Sink operation

There are two types of sink operations: native sink operation and platform sink operation. The native sink operation gets data from the platform operation when you write to a data source. The platform sink operation gets data from the native source operation when you read from a data source.

Source operation

There are two types of source operations: native source operation and platform source operation. The native source operation inputs the native data to the platform sink operation in a read projection operation. The platform source operation inputs the platform data to the native sink operation in a write projection operation.

Join operation

Use the join operation to join data from two related sources or from the same source.

Filter operation

Use the filter operation to filter data based on one or more conditions.

Projection operation

Use the projection operation to select a subset of attributes to rename or to drop fields.

For more information about interfaces that define the ASO projection model, see the *Informatica Connector Toolkit API reference documentation*.

APPENDIX C

Connector Project Migration

This appendix includes the following topics:

Connector Project Migration Overview

You can run the `ict.bat` file to migrate a connector project from Windows platform to UNIX or AIX platform.

The following table lists the available command options that you can use to migrate the connector project and deploy the connector:

Command Option	Description
`generateCode`	Generates the connector code and `build.xml` using the configurations defined in `ict_metadata.xml`.
`buildAdapter`	Compiles and deploys the connector project.
`backupProject`	Creates a backup file for the connector project.
`restoreProject`	Restores the project by using the project backup file.

The ICT command uses the following syntax:

- On Windows, `ict.bat <option> <argument1> [argument2] [...]>`
- On UNIX, LINUX and AIX, `ict.sh <option> <argument1> [argument2] [...]>`

Migrating the Adapter Project from Windows Platform to other Platforms

Before you migrate the connector project, verify that the project is available on the Windows platform.

1. Open the command prompt on the Windows platform and navigate to `%ICT_HOME%\tools\scripts`.

2. Run `ict.bat backupProject <source_dir> <destination_dir\FileName.zip>` to back up the connector project.

   Specify the following arguments:
   `source_dir`

   The source directory of your connector project .

   `destination_dir`

   The directory where you want to save the backup file.

   `FileName`

   File name for the backup file.

   The `<FileName>.zip` backup file is created in the specified directory.
3. Copy the `<FileName>.zip` file from the Windows platform to the target platform.
4. To restore the connector project, run `ict.bat restoreProject <source_dir> <destination_dir \FileName.zip>` from the target platform command prompt.

   Specify the following arguments:
   `source_dir`

   The path of your `<FileName>.zip` file.

   `destination_dir`

   The directory where you want to save the connector project.

   `FileName`

   File name for the extracted connector project.
5. Set the environment variables `ICT_HOME`, `ECLIPSE_HOME`, `JAVA_HOME` and `ANT_HOME` on the target platform.
6. Run `ict.bat generateCode <source_dir>` to generate the connector code and the `build.xml` file.

   Specify the path of the connector project in `<source_dir>`.
7. Run `ict.bat buildAdapter <source_dir>` to compile and deploy the connector project.

   Specify the path of the connector project in `<source_dir>`.

APPENDIX D

Frequently used Generic APIs

This appendix includes the following topic:

Frequently Used Generic APIs in Informatica Connector Toolkit

You can use the following generic APIs that are frequently used to create a connector using the Informatica Connector Toolkit:

Connection Adapter APIs

ConnectInfoAdapter Class

public SDKErrorInfo validateAttributes(Map<String, Object> attrNameValmap)

Validates the connection attributes and provide any validation errors.

public List<Object> getConnectInfoUpdatedConsumerInfo(Map<String, Object> map, SDKConsumerTypeEnum consumerType)

Provides the list of objects that appears in the Connection UI.

Metadata Adapter APIs

Connection Class

public Status openConnection(Map<String, Object> connAttrs)

Establishes a connection to the data source.

public Status closeConnection()

Closes the connection to the data source.

Metadata Adapter Class

Connection getMetadataConnection(List<Option> options, Map<String, Object> connAttrs)

Use this method to get the adapter metadata connection instance.

public Boolean populateObjectCatalog(Connection connection, List<Option> options, Catalog catalog)

Use this method to populate the third-party metadata information such as schemas & tables

public void populateObjectDetails(Connection connection, List<Option> options, List<ImportableObject> importableObjects, Catalog catalog)

Use this method to populate the field details & extensions for the object selected

public boolean validate(Connection sdkConnection, List<Option> options)

Use this method to validate the custom query provided by the user to fetch the metadata

public Status writeObjects(Connection connection, MetadataWriteSession writeSession, MetadataWriteOptions defOptions)

Use this method to create a data source object in the third-party to handle create target scenario

Runtime Adapter APIs

DataAdapter Class

public int initDataSession(DataSession dataSession) throws SDKException

Use this method to initialize the data session.

public int read(DataSession dataSession, ReadAttributes readAttr) throws SDKException

Use this method to implement the code to read data from the data source.

public int write(DataSession dataSession, WriteAttributes writeAttr) throws SDKException

Use this method to implement the code to write data to the data source.

public int call(DataSession dataSession, CallAttributes callAttr) throws SDKException

Use this method to implement the code to read data from the data source by executing the procedure.

public int deinitDataSession(DataSession dataSession)

Use this method to deinitialize the data session.

OperationAdapter Class

public int initDataSourceOperation(DataSourceOperationContext dsoHandle, MetadataContext connHandle) throws SDKException

Use this method to initialize the data source operation adapter before any partitions are executed.

public int deinitDataSourceOperation(DataSourceOperationContext dsoHandle, MetadataContext connHandle) throws SDKException

Use this method to deinitialize the data source operation adapter after all partitions are executed.

DataConnection Class

public Object getNativeConnection()

Use this method toreturn a data source connection object.

Plugin Class

public int initPlugin(PluginInfo pluginInfo)

This method will be called by SDK framework at the plugin loading time. Adapter developer needs to provide plugin info in this API.

public Connection createConnection()

Use this method to return adapter specific connection class instance

public DataAdapter getDataAdapter()

Use this method to return the adapter DataAdapter Object

public DataSourceOperationAdapter getDataSourceOperationAdapter()

Use this method to return the adapter DataSourceOperation object

Runtime Semantic APIs

ASOOperationObjMgr Class

public boolean validateAll(boolean recurse, ObjectManagerContext ctx, MetadataObject currentObj, MetadataObject containerObj) throws SL_ValidationException

Validates all the fields. If pushdown optimization is supported by the adapter, validate and generate pushdown SQL query for the mapping..

Typesystem APIs

Typesystem Class in the seed.provider project

DirectMapUtils.INSTANCE.createDataTypeMap(DirectTypeSystemMap directTSMap, DataType localDT, DataType foreignDT, boolean lossyToLocal, boolean lossyToForeign, boolean bestMapFromLocal, boolean bestMapFromForeign)

Creates a Connector Typesystem map with all the data types.

LicenseChecker API

LicenseChecker Class

public boolean isLicensedToRun(InformaticaLicenseChecker checker)

Validates if the user has the adapter license to execute tasks.

APPENDIX E

Frequently Asked Questions

This appendix includes the following topic:

Informatica Connector Toolkit Frequently Asked Questions

Review the frequently asked questions to answer questions you might have when you use the Informatica Connector Toolkit to develop a connector.

How to enable a Cloud Data Integration connector developed using Informatica Connector Toolkit to use the proxy server?

To enable a Cloud Data Integration connector developed using Informatica Connector Toolkit to use the proxy server, see https://kb.informatica.com/howto/6/Pages/22/564235.aspx.

How can I change the name of a connector built using Informatica Connector Toolkit?

You can change only the display name after you have developed a connector. For more information, see https://knowledge.informatica.com/s/article/How-can-I-change-the-name-of-a-connector-built-using-Informatica-Developer-toolkit?language=en_US&type=external.

INDEX