



Informatica® Development Platform

Cloud Data Integration Connector Toolkit Developer Guide

© Copyright Informatica LLC 2021, 2024

This software and documentation are provided only under a separate license agreement containing restrictions on use and disclosure. No part of this document may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording or otherwise) without prior consent of Informatica LLC.

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation is subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License.

Informatica, the Informatica logo, and Informatica Intelligent Cloud Services™ are trademarks or registered trademarks of Informatica LLC in the United States and many jurisdictions throughout the world. A current list of Informatica trademarks is available on the web at <https://www.informatica.com/trademarks.html>. Other company and product names may be trade names or trademarks of their respective owners.

Portions of this software and/or documentation are subject to copyright held by third parties. Required third party notices are included with the product.

The information in this documentation is subject to change without notice. If you find any problems in this documentation, report them to us at infa_documentation@informatica.com.

Informatica products are warranted according to the terms and conditions of the agreements under which they are provided. INFORMATICA PROVIDES THE INFORMATION IN THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND ANY WARRANTY OR CONDITION OF NON-INFRINGEMENT.

Publication Date: 2024-12-16

Table of Contents

Preface	7
Informatica Resources.	7
Informatica Network.	7
Informatica Knowledge Base.	7
Informatica Documentation.	7
Informatica Product Availability Matrices.	8
Informatica Velocity.	8
Informatica Marketplace.	8
Informatica Global Customer Support.	8
Chapter 1: Introduction to Informatica Connector Toolkit	9
Supported features.	10
Informatica Connector perspective.	11
Connector Navigator.	11
Connector Progress.	12
Chapter 2: Before you begin	13
Download the welcome kit.	13
Download and install Eclipse IDE.	13
Download and install the Informatica Connector Toolkit.	14
Install or upgrade Informatica Connector Toolkit on Windows.	14
Components of Informatica Connector Toolkit.	14
Uninstall Informatica Connector Toolkit.	15
Download OpenJDK and configure environment variables.	16
Log in to Informatica Intelligent Data Management Cloud.	17
Chapter 3: Develop a connector for Cloud Data Integration	19
Create a new connector project.	19
Phase 1: Establish a connection.	22
Create a connection.	22
Test a connection.	27
Phase 2: Define the metadata.	28
Define the data types.	29
Add metadata for JDBC driver-based connector.	30
Add metadata for third-party library-based connector.	38
Add metadata for REST API-based connector.	49
Test the metadata.	66
Package the resource or configuration files.	68
Enable metadata logger.	68
Phase 3: Configure the runtime behavior.	69

Set up runtime.	69
Test runtime behavior.	70
Create messages.	73
Implement Design-Time Messages.	74
Implement Run-Time Messages.	74
Phase 4: Generate the connector package.	75
Naming Convention.	76
Phase 5: Generate and run the test scenarios.	77
Generate the test scenarios.	78
Run the test scenarios.	82
Chapter 4: Develop a connector for Data Loader.	93
Create a new connector project.	94
Phase 1: Establish a connection.	96
Create a connection.	96
Test a connection	101
Phase 2: Define the metadata.	102
Define the data types.	102
Endpoint object metadata	103
Test the metadata.	108
Package the resource or configuration files.	110
Enable metadata logger.	110
Phase 3: Configure the runtime behavior.	111
Set up runtime.	111
Test the read capability.	111
Create messages.	113
Implement design time messages.	113
Implement runtime messages.	114
Phase 4: Generate the connector package.	114
Naming Convention-DLT.	115
Phase 5: Generate and run the test scenarios.	116
Generate the test scenarios.	117
Run the test scenarios.	121
Chapter 5: Import a connector.	132
Chapter 6: Connection attributes.	136
Connection attribute properties.	136
Chapter 7: Type system.	138
Native types and semantic categories.	138
Native Type Properties.	139
Cloud Data Integration types.	140

Chapter 8: Connector metadata.....	141
Metadata components.	141
Import dialog box settings.	142
Chapter 9: Partitioning capability.....	143
Automatic partitioning.	143
Static partitioning.	144
Chapter 10: Pushdown capability.....	145
Pushdown optimization execution flow.	145
Classes and methods for pushdown capabilities.	146
Chapter 11: Mappings in advanced mode.....	148
Project, classes, and methods.	148
Chapter 12: Manual changes to Informatica Connector Toolkit source code	154
Code changes for connection pooling.	154
Connection Pooling through API.	156
Code changes for custom query.	157
Code changes for object path override.	159
Code changes for pushdown capability.	160
Code changes for SQL transformation.	165
Code changes for stored procedures.	171
Chapter 13: Runtime behavior.....	181
Runtime Java functions.	181
Chapter 14: Connector example: MySQL_Cloud.....	183
MySQL_Cloud Connector requirements.	183
Building the sample connector.	183
MySQL_Cloud Connector components.	184
Chapter 15: Version control integration.....	186
Git version control integration.	186
Prerequisites.	186
Build a connector in Git repository	186
Perforce version control integration.	190
Prerequisites.	190
Download and install Perforce Eclipse plugin.	190
Build a connector in Perforce.	191

Appendix A: Metadata models.....	195
Metadata model components.	195
Metadata patterns.	196
Features of type A metadata template.	196
Appendix B: ASO model.....	198
ASO model components.	198
ASO projections.	199
Appendix C: Connector project migration.....	200
Migrate connector project from Windows platform to other platforms.	200
Appendix D: Frequently used generic APIs in Informatica Connector Toolkit	202
Appendix E: Frequently asked questions.....	205
Index.....	206

Preface

Use *Cloud Data Integration Informatica® Connector Toolkit Developer Guide* to learn how to develop connectors for Informatica Intelligent Cloud ServicesSM. Learn how to create connectors for different cloud services by defining the connection attributes, metadata, and runtime behavior of the connector and autogenerate code segments and test scenarios.

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 through the Informatica Network or by telephone.

To find online support resources on the Informatica Network, click **Contact Support** in the Informatica Intelligent Cloud Services Help menu to go to the **Cloud Support** page. The **Cloud Support** page includes system status information and community discussions. Log in to Informatica Network and click **Need Help** to find additional resources and to contact Informatica Global Customer Support through email.

The telephone numbers for Informatica Global Customer Support are available from the Informatica web site at <https://www.informatica.com/services-and-training/support-services/contact-us.html>.

CHAPTER 1

Introduction to Informatica Connector Toolkit

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 runtime components of the connector.
- **Deployment.** You can deploy the connector on Informatica Intelligent Cloud Services.

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

Supported features

The following table lists the features that the 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.

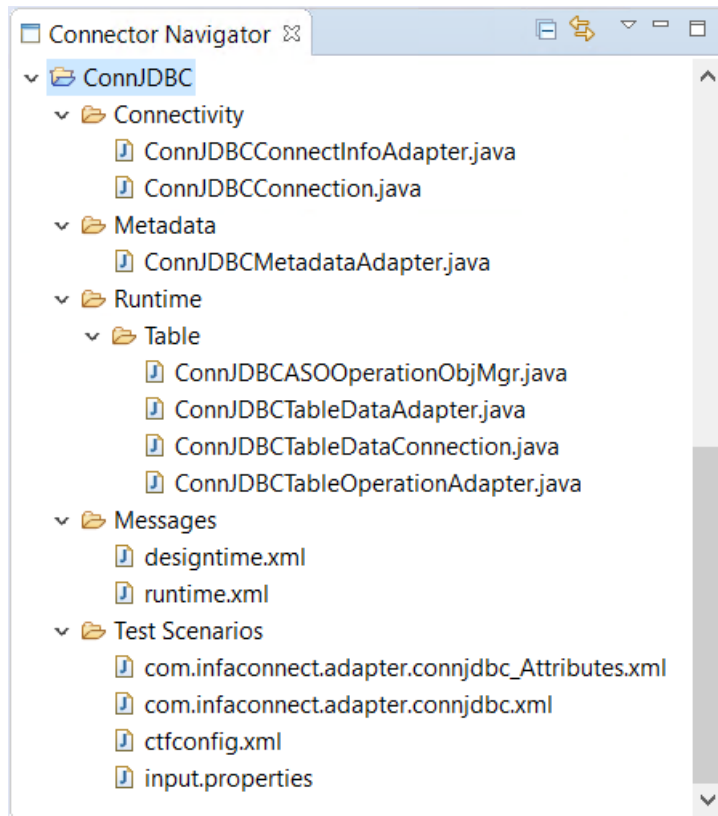
The Informatica Connector perspective consists of the Connector Navigator view and Connector Progress view in the Eclipse Workbench window, and icons in the Eclipse toolbar. You can use the Connector Navigator and Connector Progress views to define, edit, or test the connector components and view the 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 the 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.

The following image shows the source files for the five phases in the Connector Navigator view:

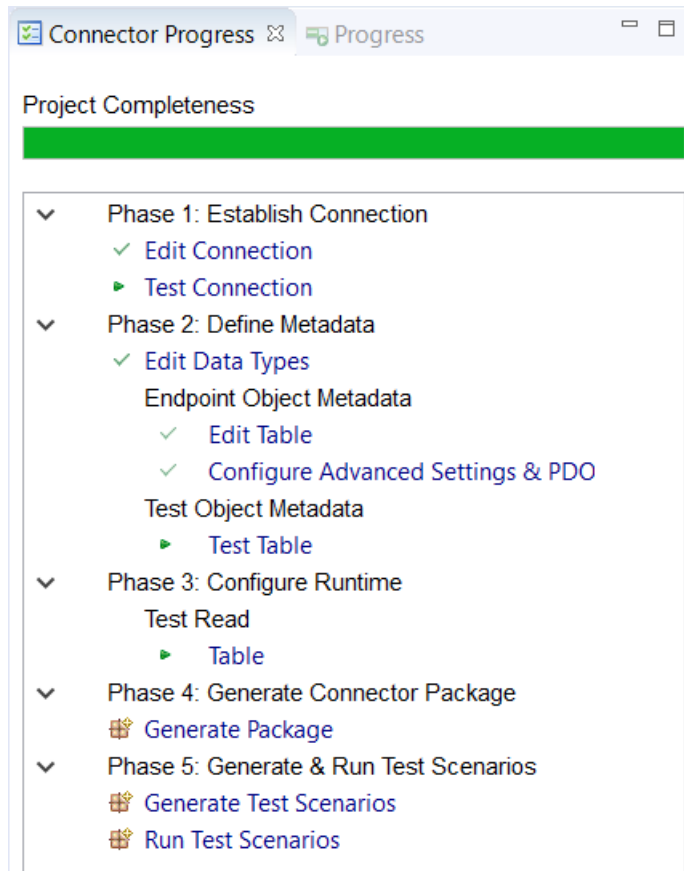


Connector Progress

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

The Connector Progress view shows progress of the project as you configure the five phases to build a connector.

The following image shows Connector Progress view with the five phases:



Building a connector involves the following phases:

1. Establish the connection.
Define, edit, or test the connection component of a connector.
2. Define the connector metadata.
Define and edit data types and endpoint object metadata for a connector. You can also test the connector metadata components.
3. Configure the runtime behavior of the connector.
Define, edit, and test the connector run-time components.
4. Generate the connector package.
Specify the location where you want to save the connector package, add a logo for the connector, and override the existing connector package.
5. Generate and run test scenarios.
Autogenerate test scenarios to quickly perform unit and integration testing for your connector.

CHAPTER 2

Before you begin

Before you develop a connector with the Informatica Connector Toolkit, complete the prerequisites.

Depending on your requirement, perform the following tasks:

1. Download the welcome kit.
2. Download and install Eclipse IDE.
3. Download and install the Informatica Connector Toolkit.
4. Download OpenJDK and configure environment variables for OpenJDK.
5. Optionally, download and install MySQL_Cloud Connector/J JDBC driver version 8.0.13 or later to use the MySQL_Cloud sample connector.
6. Log in to Informatica Intelligent Cloud Services.

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.

To download the Partner welcome kit, use the following link: [Welcome Kit](#).

Download and install Eclipse IDE

Download and install Eclipse IDE on the machine in which you plan to develop the connector.

To download Eclipse IDE for Java EE Developers with support for Plug-in Development Environment (PDE), use the following link:

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

Download Eclipse 2024-03 (4.31) that is compatible with JDK 17 to use the latest connector SDK with Eclipse IDE.

Download and install the Informatica Connector Toolkit

You can download and install the Informatica Connector Toolkit on Windows.

Download the SDK from INFACONNECT in your Informatica Intelligent Data Management Cloud account to install the Informatica Connector Toolkit. The SDK contains the installer files for the Informatica Connector Toolkit.

If you don't have an account, you can create one using the following link:

<https://infaconnect.informaticacloud.com/>

For more information on how to download the toolkit, see the [INFACONNECT documentation](#).

Install or upgrade Informatica Connector Toolkit on Windows

The Informatica Connector Toolkit installs the Informatica Connector Toolkit Eclipse plug-in and other components that you require to build a 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 or upgrade the Informatica Connector Toolkit.
5. Select the Eclipse installation directory in which you want to install or upgrade the toolkit components.
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**.
9. In the **Pre-Installation Summary** page, click **Install**.
The **Installing** page displays the installation progress.
10. Click **Done** to complete the installation, and then exit the installer.

Components of Informatica Connector Toolkit

After you install the Informatica Connector Toolkit, navigate to the installation directory to view the components of the 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. The library files of the Informatica Connector Toolkit API.
- Informatica Connector Toolkit Eclipse plug-in. The Informatica Connector Toolkit plug-in that adds the wizards and menus to the Eclipse IDE to develop, test, and deploy a connector.

- Sample MySQL_Cloud connector. The sample MySQL_Cloud Connector includes the source code that you can use as a model to build a connector.
The sample MySQL_Cloud Connector is for illustration purposes only.
- Sample source codes for the following connectors:
 - Instagram
 - MySQL
 - MySQL_Cloud
 - Redis
- Informatica Connector Toolkit API Reference. Online documentation for the Informatica Connector Toolkit API specification.

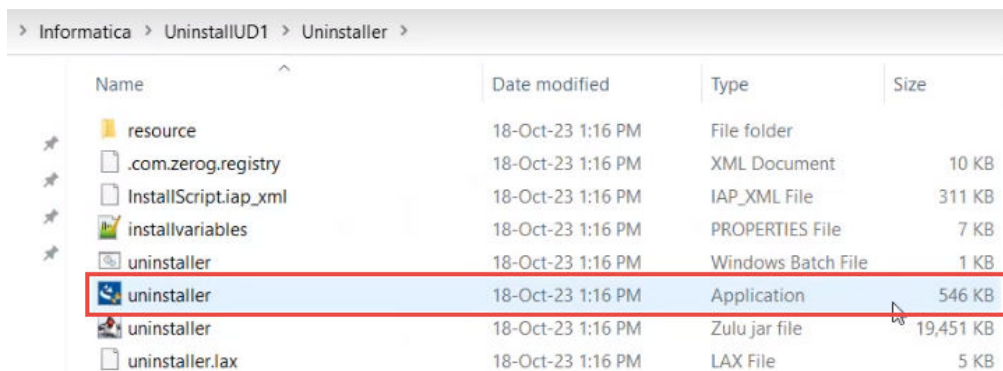
Uninstall Informatica Connector Toolkit

Uninstall Informatica Connector Toolkit to remove all the toolkit artefacts.

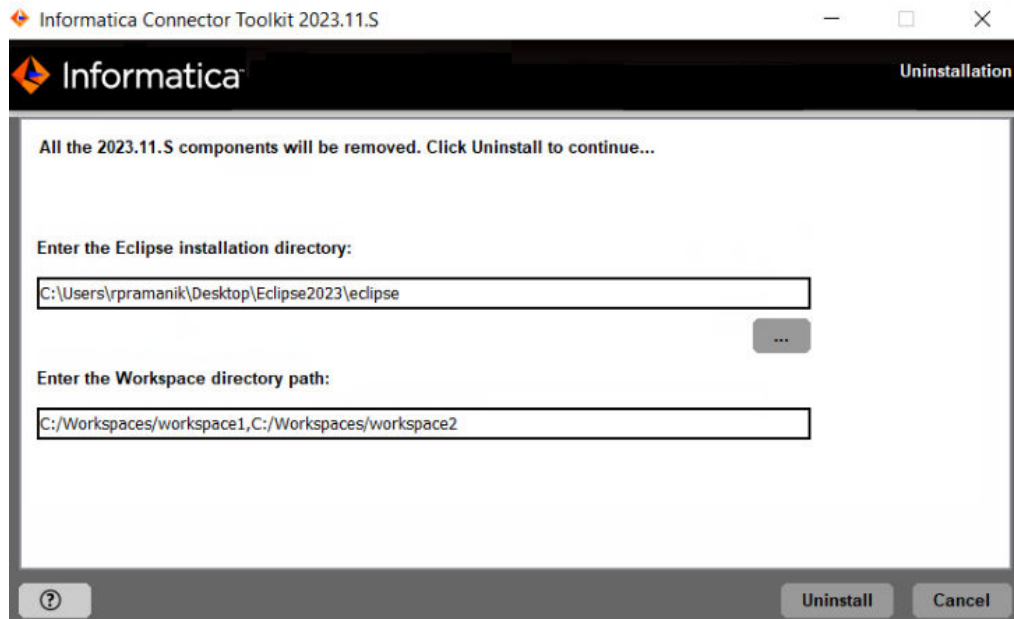
The uninstallation process deletes all the files and folders created by the installer during installation. Files and folders manually created after the installation are not removed.

Perform the following steps to uninstall the Informatica Connector Toolkit:

1. Go to the following directory:
<Installation directory>/Uninstaller
2. Run the **Uninstaller** application.



3. In the Uninstallation window, click **Uninstall**.
4. Enter the Eclipse installation directory path that contains the Informatica Connector Toolkit that you want to uninstall.



5. Enter the directory path of the workspace that you want to uninstall. You can specify multiple workspace directory paths separated by a comma.
6. After the uninstallation is complete, click **Done**.

Download OpenJDK and configure environment variables

Download and install Azul OpenJDK version 17 on Windows and configure the environment variables.

To download Azul OpenJDK, use the following link: [Download Azul JDK](#).

To configure the environment variables for Azul OpenJDK on Windows, perform the following steps:

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>.
6. Edit the Path variable and specify JAVA_HOME/bin as 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 Java installation directory or click **Directory** to navigate to the Java installation directory.

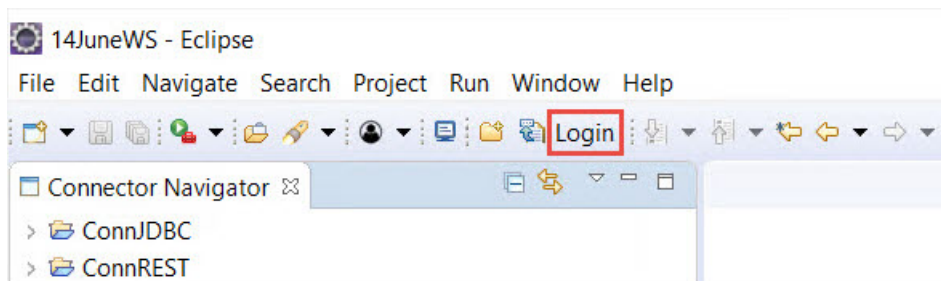
13. Enter a name for the **JRE name** property.
14. Click **Finish**.
15. Click **Apply** and then click **OK**.
16. Restart the Informatica Connector Eclipse IDE.

Log in to Informatica Intelligent Data Management Cloud

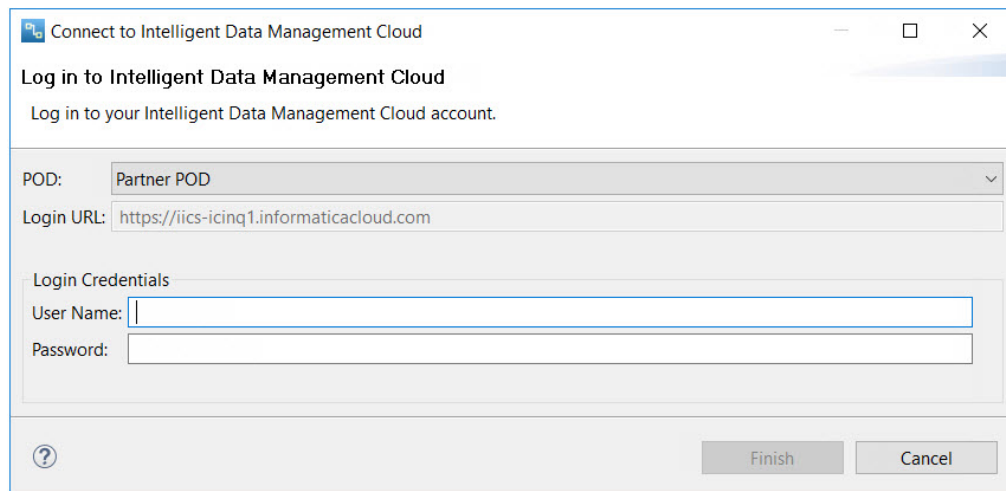
Select the POD and specify the login URL and credentials to log in to your Informatica Intelligent Data Management Cloud account.

The login window prompts you to log in if you have not logged in to your Informatica Intelligent Data Management Cloud account in the last seven days.

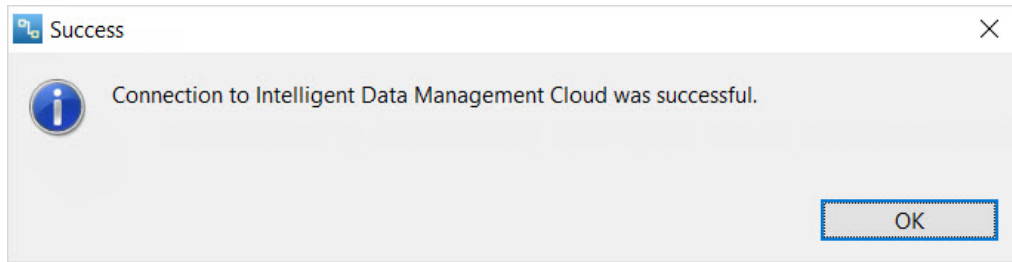
1. To trigger login, click **Login** in the Eclipse IDE toolbar.



2. In the **Connect to Intelligent Data Management Cloud** window, select the POD and specify the POD URL.



3. Specify the user name and password of your Informatica Intelligent Cloud Services account.
4. Click **Finish**.
If the authentication is successful, a success message appears.



5. Click **OK**.

CHAPTER 3

Develop 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 Cloud Data Integration.

To build a connector for Cloud Data Integration, perform the following tasks:

1. 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.
2. Define the type system for the connector. Specify the data types supported by the data source and the corresponding data types supported by Informatica.
3. Define the connector metadata, create endpoint object metadata, 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

To 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.
4. Define the connector runtime 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.
5. Deploy the connector to the Cloud Data Integration service.

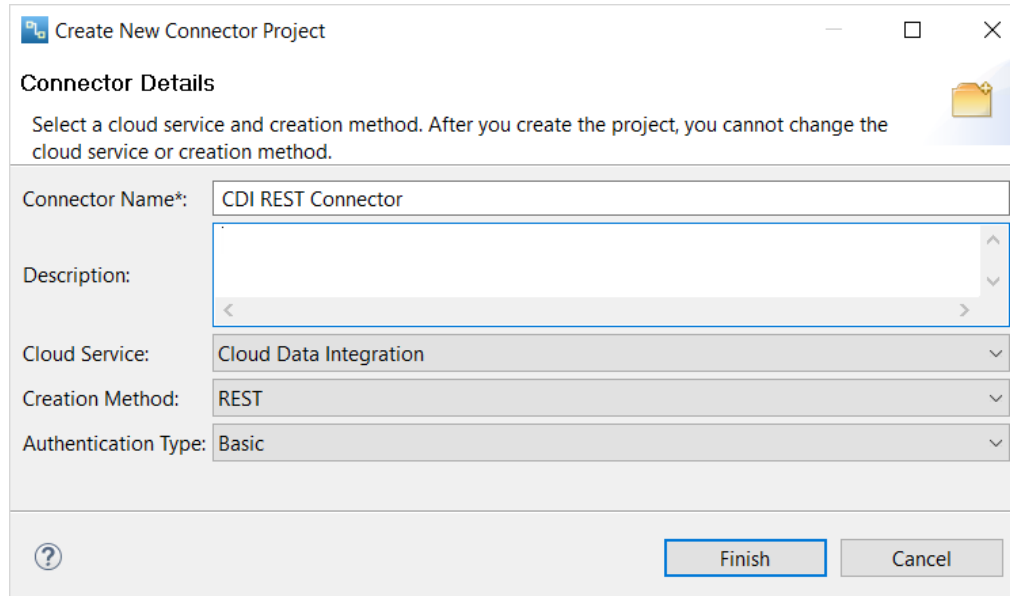
Create a new connector project

Create a connector project to define the type of connector you want to create. You can create a connector for Cloud Data Integration or Data Loader cloud service. Select the creation method, depending on whether you

want to create a connector using a JDBC driver, an SDK from a third-party library, or a REST-based solution. You can select the authentication type to connect to a REST-based application.

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

1. On the Eclipse IDE toolbar, click the **Create New Connector** button (). The **Create New Connector Project** window appears.

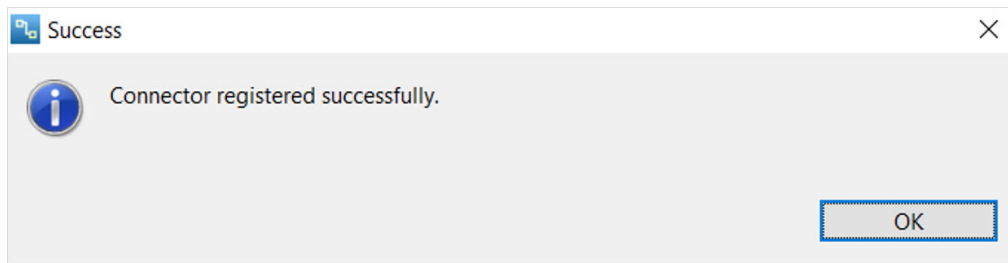


2. In the **Create New Connector Project** window, configure the following properties:

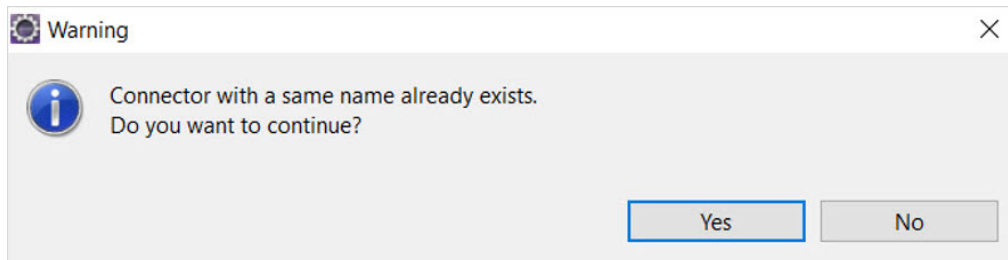
Property	Description
Connector Name	Name of the connector. Specify a unique name for the connector. The connector name must be an alphanumeric string. The first character of the name must be a letter.
Description	Description of the connector.
Cloud Service	The cloud service for which you want to create a connector. Select Cloud Data Integration .

Property	Description
Creation Method	<p>The method to create a connector.</p> <p>Select one of the following creation methods:</p> <ul style="list-style-type: none"> - JDBC. Create a JDBC driver-based connector. - Libraries. Create a connector using a third-party library SDK. - REST. Create a connector for REST-based application. <p>Default is REST.</p>
Authentication Type	<p>The authentication type for the REST creation method.</p> <p>Select one of the following authentication types:</p> <ul style="list-style-type: none"> - No Authentication: Does not require authentication. - Basic. Uses the 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. <p>Default is basic authentication.</p>

3. If a connector with the same name already exists, the following error is displayed:
A connector with the same name already exists. Enter a different name.
Click **OK** and specify a unique name for the new connector.
4. Click **Finish**.
If the new connector is successfully registered, a success message appears.



5. Click **OK**.
The new connector project appears on the **Connector Navigator** tab.
6. If the connector is already registered in the same organization, a warning appears. Click **Yes** to skip the registration process.



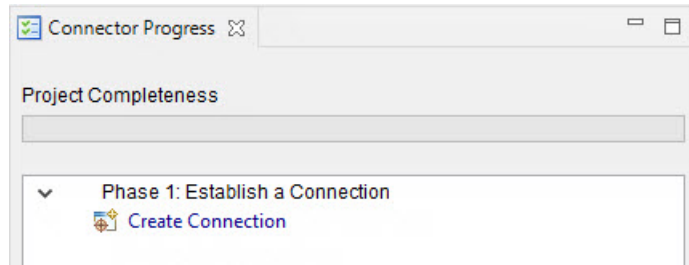
Phase 1: Establish a connection

Use the Informatica Connector Toolkit to define the connection attributes, create connection pages, and add third-party libraries required to connect to the data source.

Create a connection

Define the connection attributes to authenticate access to a data source. Create a connection page to define the attributes to authenticate access to a data source. Organize attributes into specific sections, grouping similar ones. Include the required third-party libraries to set up the connection.

1. In the **Connector Progress** view, under **Phase 1: Establish Connection**, click **Create Connection**.



2. Configure the connection attributes.

Phase 1: Establish Connection > Create

Connection Attributes

Use the default connection attributes or add, edit, or delete attributes of your choice.

Name	Display Na...	Data Type	Mandatory	Encrypted	Has Depen...
baseURL	Base URL	String	✓	✗	✗
username	username	String	✓	✗	✗
password	password	String	✓	✓	✗

Buttons: Add, Edit, Delete

Navigation: ? < Back, Next >, Save, Generate Code, Cancel

- a. Click **Add** to add a new connection attribute.
The attributes that you configure correspond to the connection object properties.

- b. In the **Add Attribute** window, 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.
Supports Parameter	Indicates whether you can parameterize 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.
Has Dependent Fields	Indicates whether the attribute has dependent fields.
Allowed Values	List of values allowed for the attribute.

- c. Click **OK**.
3. Click **Next**.
4. Configure the connection pages.

Phase 1: Establish Connection > Create

Connection Pages

Create sections in the connection page and add the attributes to specific sections. You can group similar attributes under a section.

Attributes:

>>
>
<
<<

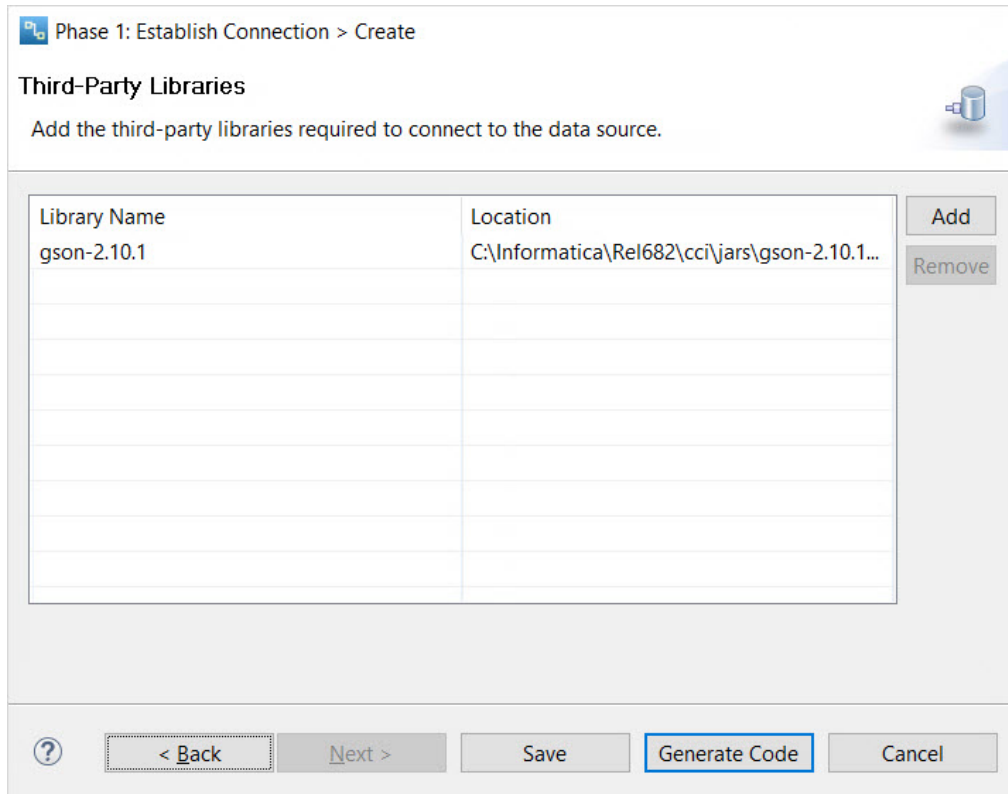
Connection Pages:

- ▼ Connection Page
 - ▼ Connection Section
 - baseURL
 - username
 - password

Add Section
Edit
Remove
Move Up
Move Down

?
< Back
Next >
Save
Generate Code
Cancel

- a. To create a new connection section, click **Add Section** and enter the section name, section title, tool tip, and section description.
- b. Add attributes to specific sections. You can group similar attributes under a section.
- c. To change the order in which connection sections appear in a connection page, use **Move Up** or **Move Down**.
5. Click **Next**.
6. Add the third-party libraries.
Click **Add** to add the third-party libraries required to connect to the data source.



7. Click **Generate Code**.
After you define the connection attributes, the Informatica Connector Toolkit generates the following Java files:

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

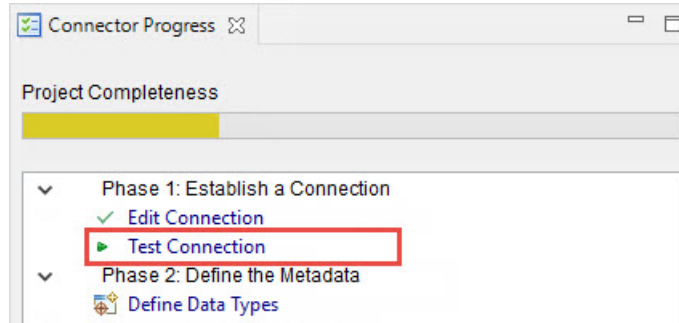
8. Update the `ConnectInfoAdapter.java` file to implement connection validation and attribute dependencies. Also, update the `Connection.java` file to implement the methods that open and close connection to the data source.
9. You can test and debug the connection to the data source.

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.

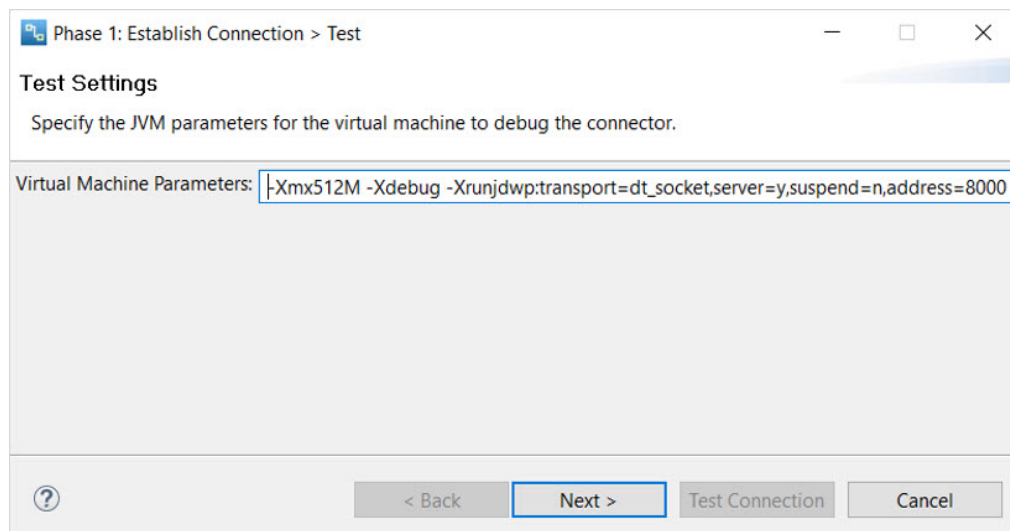
Test a connection

Specify the JVM parameters and connection details to test and debug the connection to the data source.

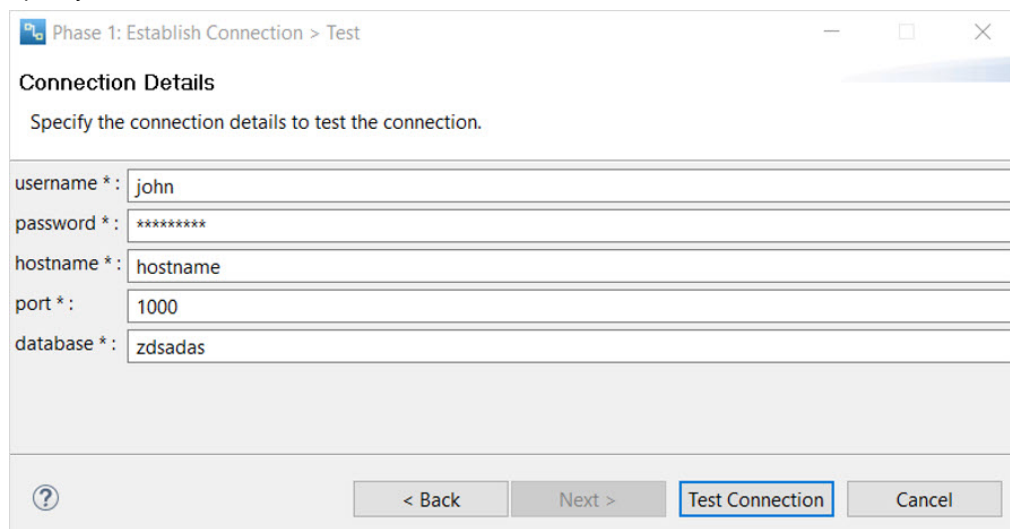
1. Click **Test Connection**.



2. Specify the JVM parameters for the virtual machine to debug the connector.



3. Click **Next**.
4. Specify the connection details to test the connection.



5. Click **Test Connection**.

Phase 2: Define the 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

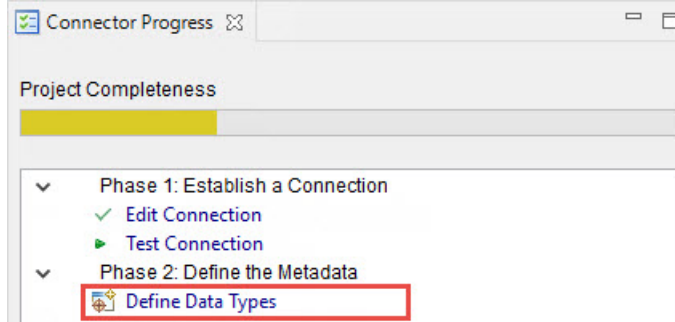
`<ICT installation directory>/CCI/plugins/infra` to `<ICT installation directory>/ICT/ Application/plugins` and restart the Informatica Connector Toolkit.

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.

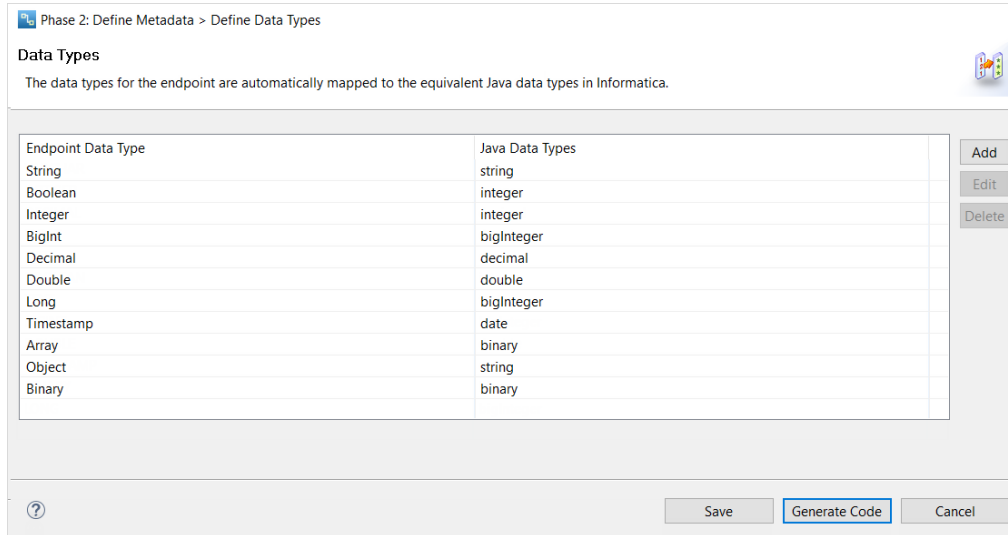
Define the data types

The data types for the endpoint are automatically mapped to the equivalent Java data types in Informatica. You can add, edit, or delete data types according to your requirement.

1. In the **Connector Progress** view, under **Phase 2: Define Metadata**, click **Define Data Types**.



The data types for the endpoint are automatically mapped to the equivalent Java data types in Informatica.



2. Perform the following steps to add a data type:
 - a. Click **Add**.
 - b. In the **Add Data Type** window, configure the following properties for data type:

Property	Value
Type Name	Name for the data type.
Comments	Comments for the data type.

Property	Value
Best Java Type for Read	The Java data type in Informatica that best maps to the endpoint data type when the connector reads from the data source.
Properties	<p>Set one or more of the following properties for the data type:</p> <ul style="list-style-type: none"> - 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. <p>The maximum length of any attribute cannot be greater than Integer.MAX_VALUE, which is 2,147,483,647.</p> <ul style="list-style-type: none"> - Unit of length such as characters, bytes, and bits. - Date properties such as hour, minute, second, year, month, day, and time zone.

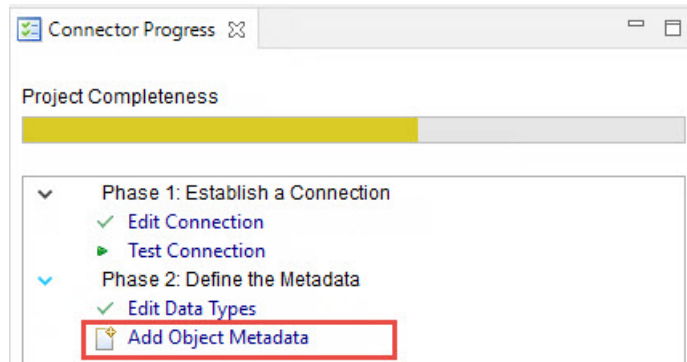
c. Click **OK**.

3. Click **Generate Code**.

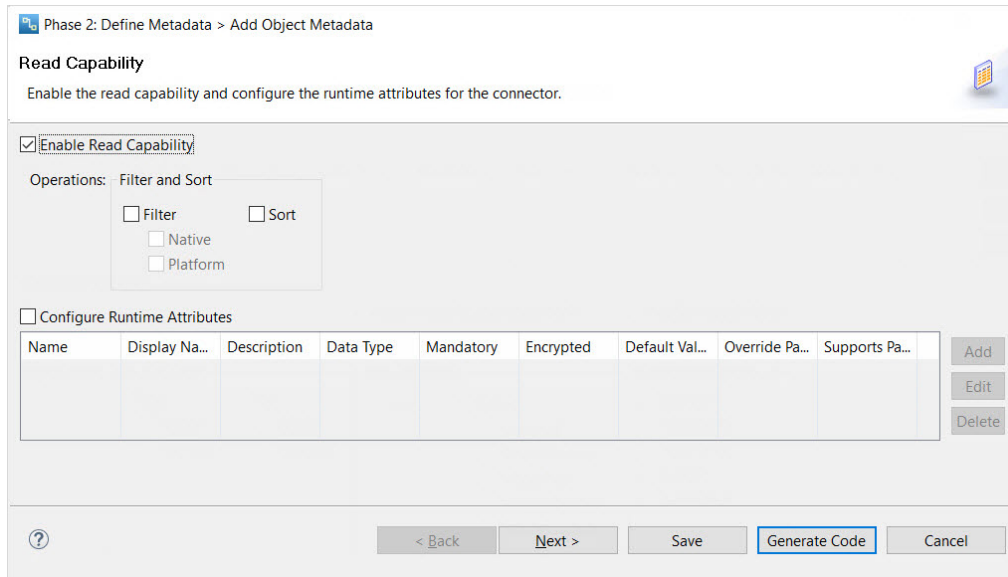
Add metadata for JDBC driver-based connector

Define the object metadata and enable the read and write capabilities for a JDBC driver-based connector

1. Click **Add Object Metadata**.



2. To enable read capability for the connector, select **Enable Read Capability**.



- a. Select whether the connector supports filter operation when the connector reads from the data source.
 - To specify operators and expression syntax recognized by Cloud Data Integration for the filter operation, select **Platform**.
 - 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**.

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 the `MySQL_Cloud` sample connector in the following location:

```
source\ict\samples\MySQL_Cloud
```

- b. Select whether the connector supports sort to retrieve data from the data source in a specific order.
- c. To add runtime attributes, select **Configure Runtime Attributes**, and then click **Add** and configure the following attribute properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the connection attribute.
Data Type	Data type of the attribute.
Default Value	Default value for the attribute.

Property	Description
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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
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.
Override Partitions	Indicates if the attribute can be overridden for each partition. Implement the <ConnectorID><NMOName>AutoPartitioningMetadataAdapter file to define the partition support.
Allowed Values	List of values allowed for the attribute.

- d. Click **OK**.
3. Click **Next**.
4. To define write capability for the connector, select **Enable Write Capability**.

Phase 2: Define Metadata > Add Object Metadata

Write Capability

Enable the write capability and configure the runtime attributes for the connector.

Enable Write Capability

Write Operations:

Insert Update Delete

Upsert

Configure Runtime Attributes

Name	Display Na...	Description	Data Type	Mandatory	Encrypted	Default Val...	Override Pa...	Supports Pa...

Add Edit Delete

< Back Next > Save Generate Code Cancel

a. Select the write operations that the connector supports.

- Insert
- Update
- Delete
- Upsert

When you select Upsert, 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.

b. To add runtime attributes, select **Configure Runtime Attributes**, and then click **Add** and configure the following attribute properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the attribute.
Default Value	Default value for the attribute.

Property	Description
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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
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.
Override Partitions	Indicates if the attribute can be overridden for each partition. Implement the <ConnectorID><NMOName>AutoPartitioningMetadataAdapter file to define the partition support.
Allowed Values	List of values allowed for the attribute.

c. Click **OK**.

5. 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.

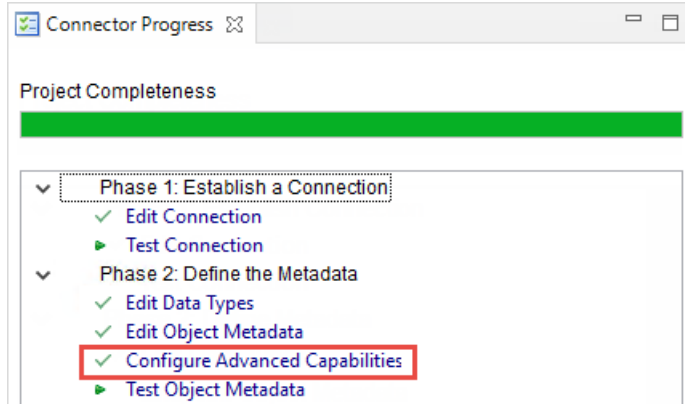
If you configured metadata write settings for the connector, implement the writeObjects method in the <NMOName>MetadataAdapter.java file.

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.

Configure advanced capabilities

Configure the advanced read and write capabilities for a JDBC driver-based connector. Enable the record extension and field extension to include additional metadata information for objects and fields.

1. Click **Configure Advanced Capabilities**.



2. Enable the advanced read capabilities for the connector.

Phase 2: Define Metadata > Configure Advanced Capabilities

Advanced Read Capabilities

Enable the advanced read capabilities.

Feature Support:	Operations:	Select	Expression Operators
<input type="checkbox"/> Lookup	<input type="checkbox"/> Join	<input type="checkbox"/> Select	<input type="checkbox"/> Contains
<input type="checkbox"/> Foreign Key	<input type="checkbox"/> Native	<input type="checkbox"/> Select All	<input type="checkbox"/> Starts With
<input type="checkbox"/> Custom Query	<input type="checkbox"/> Platform	<input type="checkbox"/> Select Distinct	<input type="checkbox"/> Ends With
<input type="checkbox"/> Object Path Override		<input type="checkbox"/> Select Any	<input type="checkbox"/> Is Null
<input type="checkbox"/> Partitioning		<input type="checkbox"/> Select First Row	<input type="checkbox"/> Is Not Null
<input type="checkbox"/> Fixed		<input type="checkbox"/> Select Last Row	
<input type="checkbox"/> Key Range			

- a. Select whether the connector supports lookup, foreign key, custom query, object path override, and partitioning when the connector reads from the data source.
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.
- b. Select whether the connector supports join operation when the connector reads from the data source.
 - To specify operators and expression syntax recognized by Cloud Data Integration for the join operation, select **Platform**.

- To specify an expression for the join operation that is specific to the data source for which you build the connector, select **Native**.

Note: When a connector supports platform join, you can use the default operators such as =, !=, >, >=, <, and <= for join conditions. You can specify advanced operators such as Contains, Starts With, Ends With, Is Null, and Is Not Null for join 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 the MySQL_Cloud sample connector in the following location:

```
source\ict\samples\MySQL_Cloud
```

- Select whether the connector supports Select operation to retrieve data for select rows from the data source.
- Click **Next**.
 - To add additional metadata information for the object and fields, select **Add Record Extension** and **Add Field Extension**.

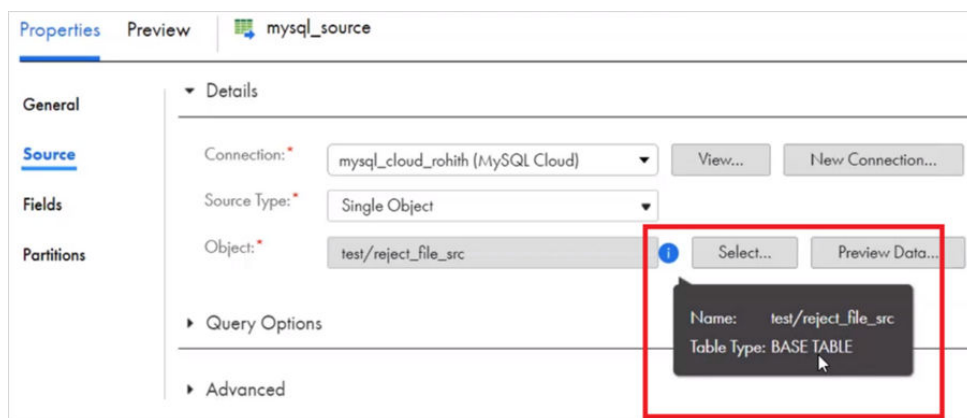
- Click **Add** to add a record extension or field extension and configure the following properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the attribute.

Property	Description
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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
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.

b. Click **OK**.

The following shows an example of record extension in Cloud Data Integration:



The following image shows an example of field extension in Cloud Data Integration:



5. 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()

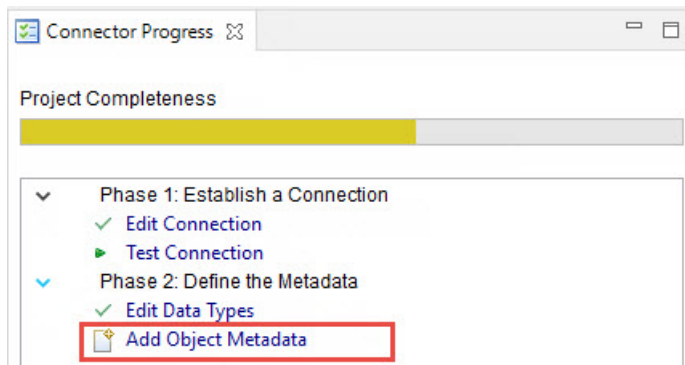
Gets metadata from the data source based on the import dialog options settings.

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

Add metadata for third-party library-based connector

Define the object metadata for a connector created using a third-party library SDK. Enable record extension and field extension to add additional metadata information for the object and fields respectively. Also, enable read and write capabilities for the connector.

1. Click **Add Object Metadata** under Endpoint Object Metadata.



2. Enter the endpoint object metadata details.

Property	Description
Name	Name of the object.
Display Name	Display name of the object.
Description	Description of the object.
Custom Query	Select to use a custom query to import a data source. When you select Custom Query , you can select Query as the source type to import a data source.
Object Path Override	Select to override the object path at runtime.

The following image shows the endpoint object metadata details:

Phase 2: Define Metadata > Add Object Metadata

Endpoint Object Metadata

Enter a unique name for the endpoint metadata object. You cannot change the name after you have defined the endpoint metadata definition.

Name:

Display Name:

Description:

Pattern Type: Record

Custom Query

Object Path Override

Use Existing Runtime

? < Back Next > Save Generate Code Cancel

3. Click **Next**.
4. To add additional metadata information for the object, select **Add Record Extension**.

Phase 2: Define Metadata > Add Object Metadata

Endpoint Object Definition

Enter the endpoint metadata record details.

Add Record Extension

Name	Display Na...	Description	Data Type	Mandatory	Encrypted	Default Val...

- a. Click **Add** to add a record extension and configure the following properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the 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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.

Property	Description
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.

- b. Click **OK**.
5. Click **Next**.
6. To add additional metadata information for the fields, select **Add Field Extension**.

- a. Click **Add** to add a field extension and configure the following properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.

Property	Description
Data Type	Data type of the 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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
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.

- b. Click **OK**.
7. Click **Next**.
8. To enable read capability for the connector, select **Enable Read Capability**.

Read Capability

Enable the read capability and configure the runtime attributes for the connector.

Enable Read Capability
 Configure Runtime Attributes

Name	Display Na...	Description	Data Type	Mandatory	Encrypted	Default Val...	Override Pa...	Supports Pa...	

Buttons: Add, Edit, Delete

Feature Support:

Lookup: Required Enable read in advanced mode Read using Bulk API
 Not Required

Operations:

Join and Filter: Filter Join
 Native Native
 Platform Platform

Sort and Select: Sort Select
 Select All
 Select Distinct
 Select Any
 Select First Row
 Select Last Row

Expression Operators: Contains
 Starts With
 Ends With
 Is Null
 Is Not Null

Buttons: ? < Back Next > Save Generate Code Cancel

- a. To add runtime attributes, select **Configure Runtime Attributes**, and then click **Add** and configure the following attribute properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the connection attribute.
Data Type	Data type of the 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.

Property	Description
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
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.
Override Partitions	Indicates if the attribute can be overridden for each partition. Implement the <ConnectorID><NMOName>AutoPartitioningMetadataAdapter file to define the partition support.
Allowed Values	List of values allowed for the attribute.

- b. Select whether the connector supports lookup of data when the connector reads from the data source.
- c. Select **Enable read in advanced mode** to enable read capability for mappings in advanced mode. For more information, see [Chapter 11, "Mappings in advanced mode" on page 148](#).
- d. Select **Read using Bulk API** if the connector supports bulk processing to read large amount of data from the source.
- e. 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
```

- f. Select whether the connector supports sort operation to retrieve data from the data source in a specific order.

- g. Select whether the connector supports select operation to retrieve data for select rows from the data source.
9. Click **Next**.
 10. To define write capability for the connector, select **Enable Write Capability**.

Phase 2: Define Metadata > Add Object Metadata

Write Capability
 Enable the write capability and configure the runtime attributes for the connector.

Enable Write Capability
 Configure Runtime Attributes

Name	Display Na...	Description	Data Type	Mandatory	Encrypted	Default Val...	Override Pa...	Supports Pa...

Write Operations:
 Insert Update Delete
 Upsert Enable Write Bulk API Enable write in advanced mode

< Back Next > Save **Generate Code** Cancel

- a. To add runtime attributes, select **Configure Runtime Attributes**, and then click **Add** and configure the following attribute properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the 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.

Property	Description
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
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.
Override Partitions	Indicates if the attribute can be overridden for each partition. Implement the <ConnectorID><NMOName>AutoPartitioningMetadataAdapter file to define the partition support.
Allowed Values	List of values allowed for the attribute.

b. Select the write operations that the connector supports.

- Insert
- Update
- Delete
- Upsert

When you select Upsert, 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.

c. 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.

d. Select **Enable write in advanced mode** to enable write capability for mappings in advanced mode. For more information, see [Chapter 11, "Mappings in advanced mode" on page 148](#).

11. Click **Next**.

12. To define pushdown capability for the connector, select **Enable Pushdown Capability**.

Phase 2: Define Metadata > Add Object Metadata

Pushdown Capability for the Endpoint Metadata Object

Add atleast one pdo attribute to proceed

Enable Pushdown Capability

Adapter ID	Supports Source Pushdown	Supports MultiTarget	Supports Si...	Supports C...

Supported transformations for pushdown

Filter Joiner Sorter
 Expression Aggregator Lookup
 Union Router Sequence Generator
 Update Strategy SQL

< Back Next > Save **Generate Code** Cancel

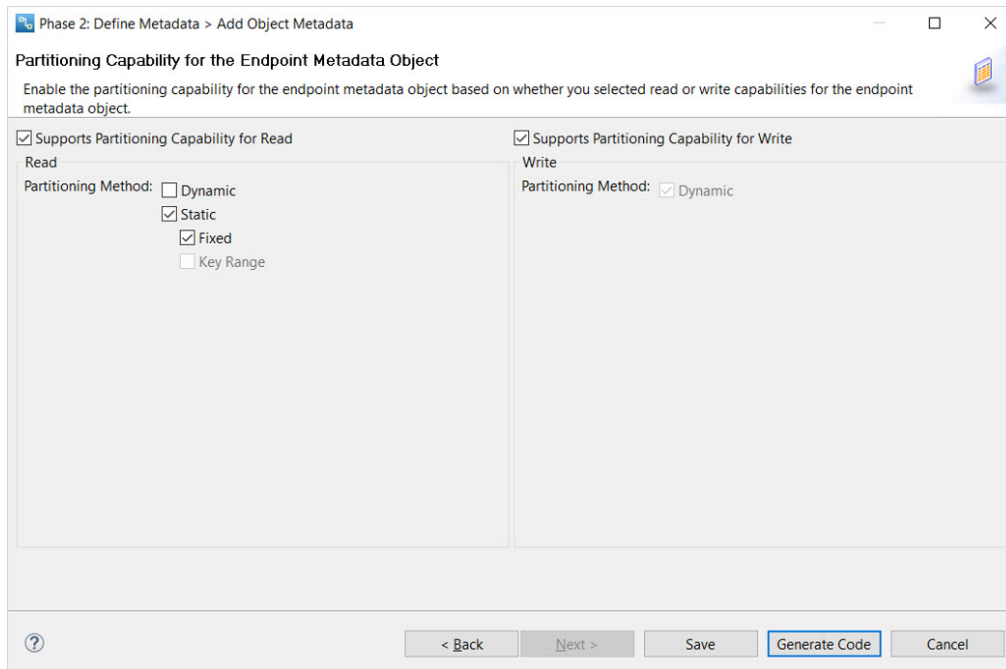
When you specify the pushdown attribute properties, you can define the adapter ID for the source. Define whether the connector supports source or full pushdown or pushdown to single or multiple targets. When you enable pushdown capability, you must add at least one source adapter ID.

Select the transformations that the connector must support for pushdown optimization:

- Filter
- Joiner
- Sorter
- Expression
- Aggregator
- Lookup
- Router
- Union
- Update Strategy
- Sequence Generator

13. Click **Next**.

14. Select whether the connector supports partitioning capability for the read operation.



- a. 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.
 - b. To configure the connector to get partition information from the user, select the **Static** partitioning method.
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.
 - c. 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.
15. 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.

Add metadata for REST API-based connector

Define the object metadata for a connector created for a REST-based application. You can select the flattened or hierarchical output format type for the object.

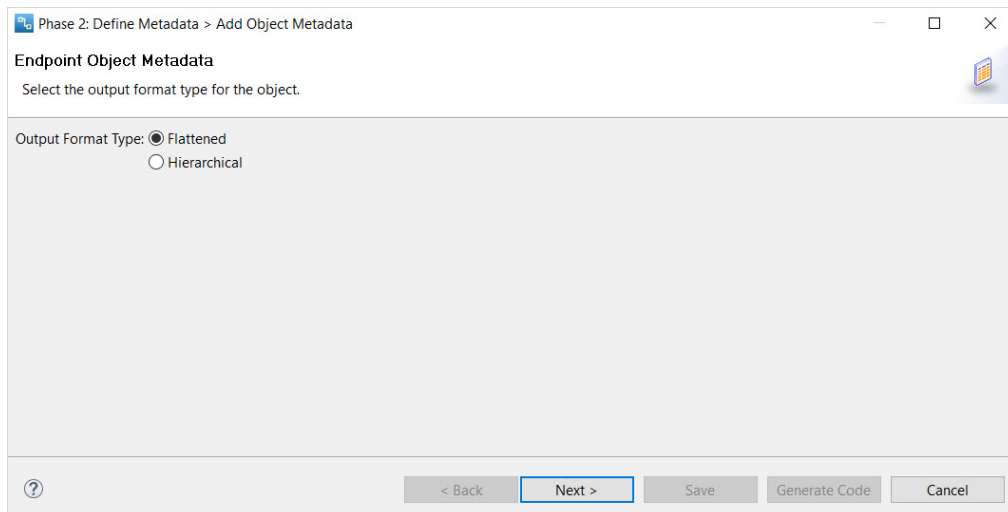
Flattened output

Configure the metadata of the object of flattened type. Also, enable the read and write capabilities for the connector.

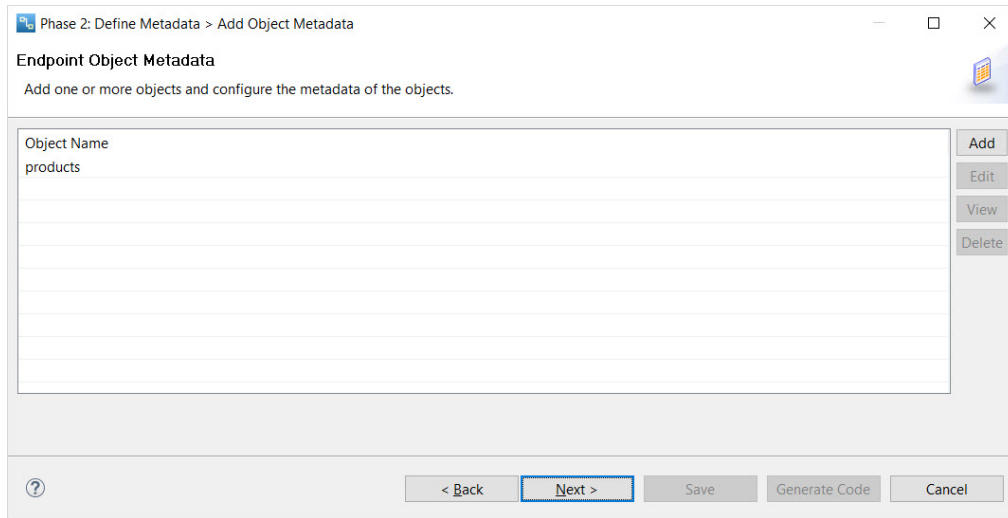
1. Click **Add Object Metadata**.



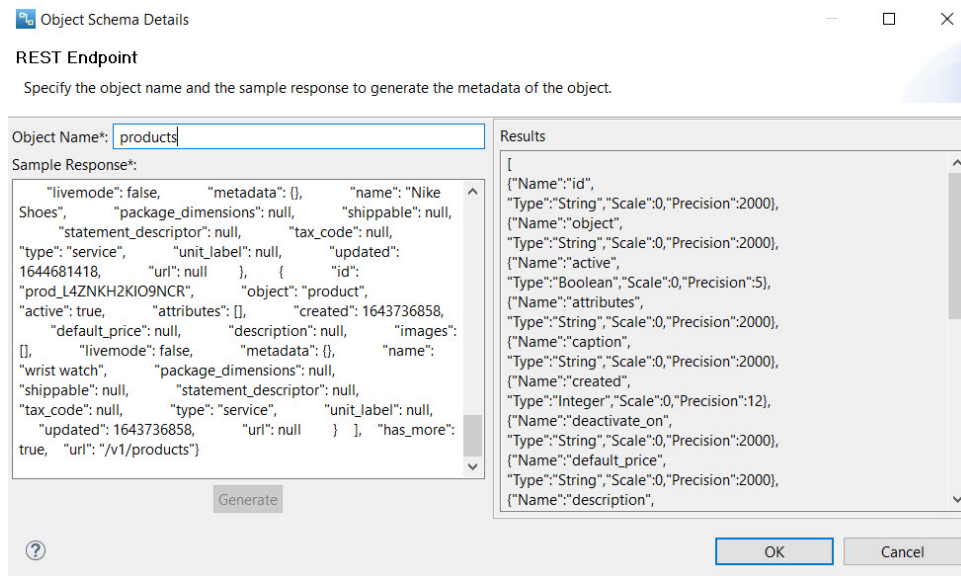
2. Select **Flattened** as output format type.



3. Add one or more objects and configure the metadata of the objects.



- a. Click **Add** to add an object and generate the metadata of the object.
- b. Specify the **Object Name** and **Sample Response**.



- c. Click **Generate** to generate the metadata of the object.
- d. Click **OK**.
4. Click **Next**.
5. To enable the read capability for the connector, select **Enable Read Capability**.

Phase 2: Define Metadata > Add Object Metadata

Read Capability

Enable the read capability and configure the runtime attributes for the connector.

Enable Read Capability
 Configure Runtime Attributes

Name	Display Na...	Description	Data Type	Mandatory	Encrypted	Default Val...	Override Pa...	Support

Add
Edit
Delete

- a. To add runtime attributes, select **Configure Runtime Attributes**, and then click **Add** and configure the following attribute properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the 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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
Encrypted	Indicates whether you can encrypt the 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.
Override Partitions	Indicates if the attribute can be overridden for each partition. Implement the <ConnectorID><NMOName>AutoPartitioningMetadataAdapter file to define the partition support.
Allowed Values	List of values allowed for the attribute.

- b. Click **OK**.
6. Click **Next**.
7. To enable the write capability for the connector, select **Enable Write Capability**.

- a. To add runtime attributes, select **Configure Runtime Attributes**, and then click **Add** and configure the following attribute properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.

Property	Description
Data Type	Data type of the 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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
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.
Override Partitions	Indicates if the attribute can be overridden for each partition. Implement the <ConnectorID><NMONName>AutoPartitioningMetadataAdapter file to define the partition support.
Allowed Values	List of values allowed for the attribute.

b. Click **OK**.

8. Click **Generate Code**.

After you define the connector metadata, the Informatica Connector Toolkit generates the <NMONName>MetadataAdapter.java file in the **Metadata** folder. Implement the following methods in the <NMONName>MetadataAdapter.java file to import metadata.

populateObjectCatalog()

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

Note: Data preview 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 <NMONName>MetadataAdapter.java file.

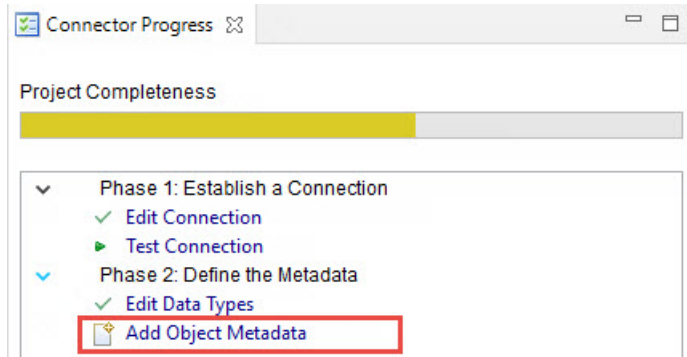
Note: If you regenerate code for the endpoint metadata definition project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica Connector perspective.

You have to manually edit the source code and make changes if you add, remove, or change the endpoint metadata attributes.

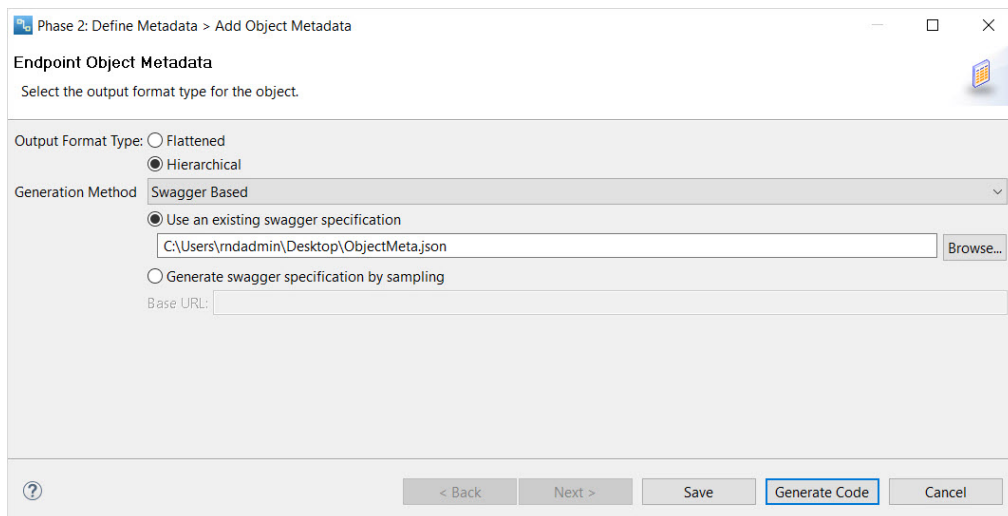
Hierarchical output

Configure the metadata of the object of hierarchical type. Use the Swagger-based or API based method to generate the metadata for an object.

1. Click **Add Object Metadata**.



2. Select **Output Format Type** as **Hierarchical**.



3. Select one of the following generation methods:

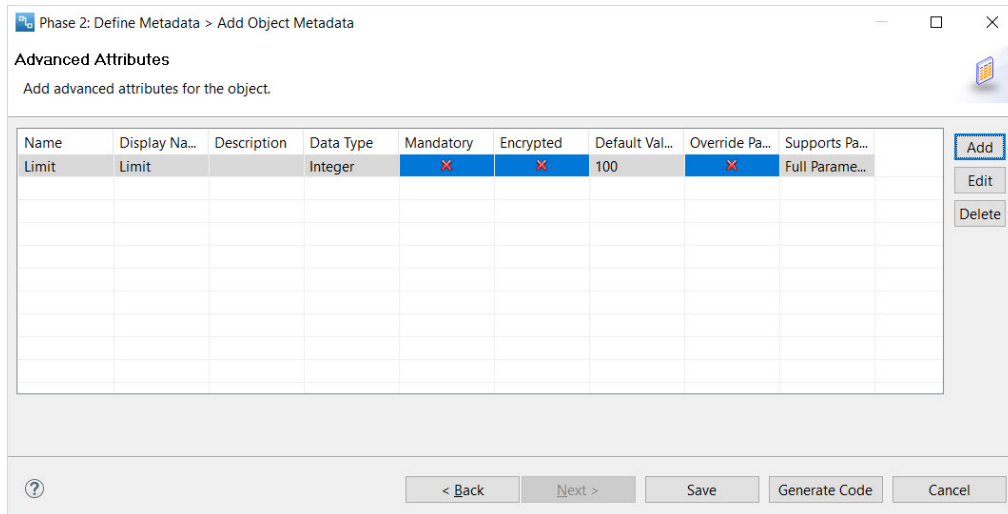
- Swagger Based
- API Based

4. Click **Next**.

API-based

Use the API-based method to generate metadata for an object of hierarchical type.

1. Add advanced attributes for the object.

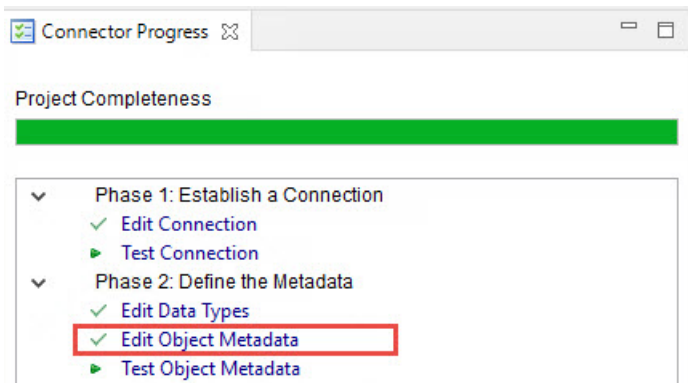


- a. Click **Add** and configure the following fields to add advanced attributes for the object.

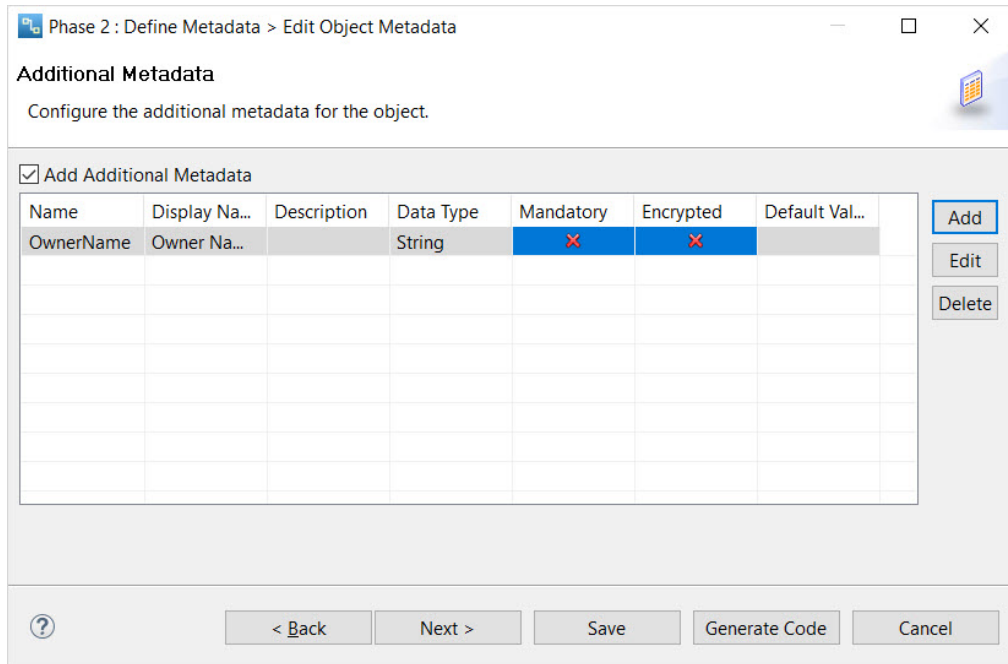
Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the attribute.
Default Value	Default value for the attribute.
Min Length	Minimum length for the value of the attribute.
Max Length	Maximum length 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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
Encrypted	Indicates whether you can encrypt the 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.
Override Partitions	Indicates if the attribute can be overridden for each partition. Implement the <ConnectorID><NMOName>AutoPartitioningMetadataAdapter file to define the partition support.
Allowed Values	List of values allowed for the attribute.

- b. Click **OK**.
2. Click **Generate Code**.
Note: The first time you add metadata for API-Based generation method, you can only add the advanced attributes for the object. To configure the additional metadata and additional field information, you must edit the object.
3. Click **Edit Object Metadata**.



4. Click **Next**.
5. Select **Add Additional Metadata** to add additional information for the object.



a. Click **Add** and configure the following fields to add additional metadata for the object.

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the 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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - No. The attribute does not support parameterization.
Encrypted	Indicates whether you can encrypt the 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.

- b. Click **OK**.
6. Click **Next**.
7. Select **Add Additional Field Information** to add additional information for the fields.

- a. Click **Add** and configure the following fields to add additional metadata for the fields.

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the attribute.

Property	Description
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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - No. The attribute does not support parameterization.
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.

b. Click **OK**.

8. 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 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.

Swagger-based

When you use the Swagger-based method to generate metadata for an object of hierarchical type, you can use one of the following options:

Use an existing swagger specification

You can select an existing Swagger 2.0 file or an Open API file to generate the metadata.

1. Select **Use an existing swagger specification** and click **Browse** to upload a swagger file in JSON format.
2. Click **Generate Code**.

Generate swagger specification by sampling

You can use a sample response to generate the metadata. When you use a sample response, the toolkit generates a Swagger 2.0 file.

1. Select **Generate swagger specification by sampling** and specify the **Base URL**.

Phase 2: Define Metadata > Add Object Metadata

Endpoint Object Metadata
Select the output format type for the object.

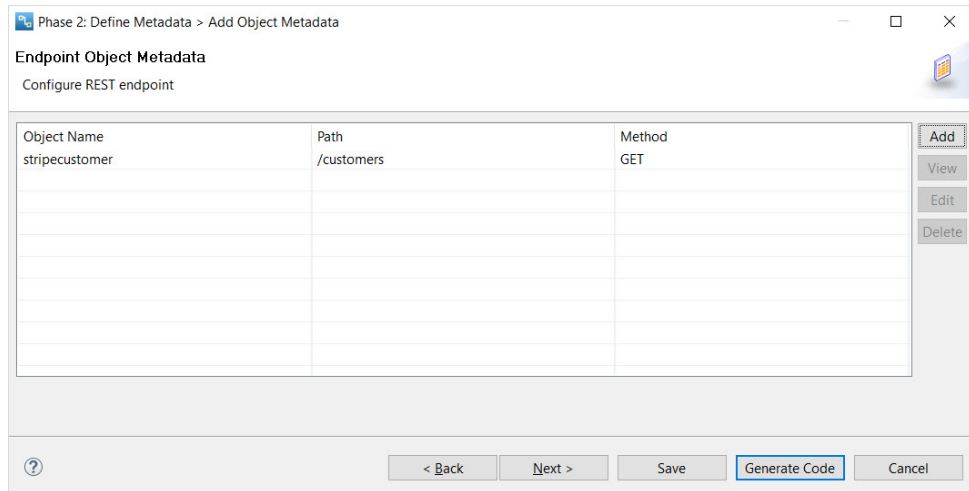
Output Format Type: Flattened
 Hierarchical

Generation Method: ▾

Use an existing swagger specification
C:\Users\rndadmin\Desktop\ObjectMeta.json

Generate swagger specification by sampling
Base URL:

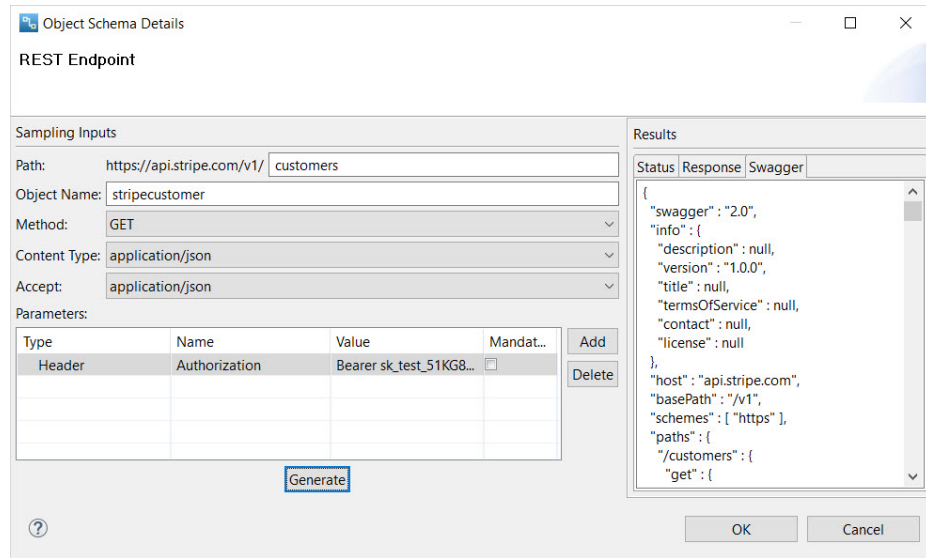
2. Click **Next**.
3. Configure the REST Endpoint.



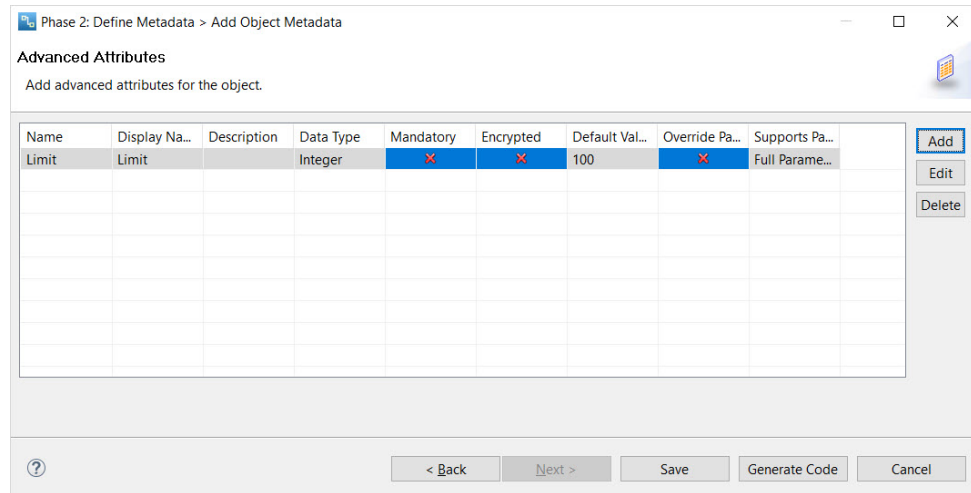
- a. Click **Add** and configure the following fields to generate the swagger specification for the REST endpoint:

Property	Description
Path	The path of the REST endpoint.
Object Name	Name of the object.
Method	The HTTP method used to retrieve data from or send data to the endpoint.
Content Type	The content type header to specify the content type of the request body. <ul style="list-style-type: none"> - application/json. Indicates that the request body format is JSON. - application/xml. Indicates that the request body format is XML. - application/x-www-form-urlencoded. Indicates that the request body is URL encoded.
Accept	The accept header to specify the content type of the output. <ul style="list-style-type: none"> - application/json. Indicates that the output type is JSON. - application/xml. Indicates that the output type is XML.

The following image shows the field you must configure to generate the swagger specification:



- b. Click **Generate**.
 - c. Click **OK**.
4. Click **Next**.
 5. Add advanced attributes for the object.



- a. Click **Add** and configure the following fields to add advanced attributes for the object.

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.

Property	Description
Data Type	Data type of the 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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
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.
Override Partitions	Indicates if the attribute can be overridden for each partition. Implement the <code><ConnectorID><NMOName>AutoPartitioningMetadataAdapter</code> file to define the partition support.
Allowed Values	List of values allowed for the attribute.

b. Click **OK**.

6. 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 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.

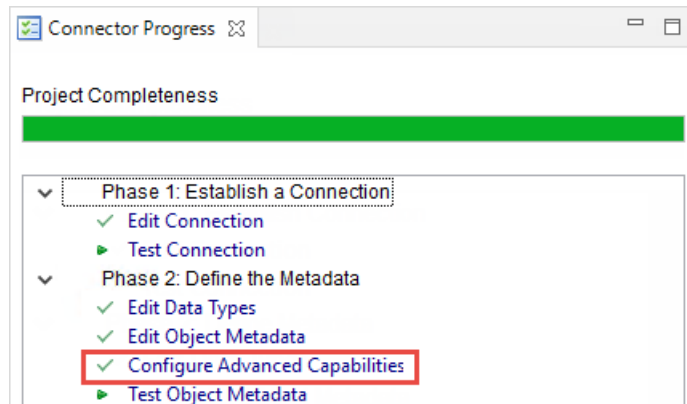
Note: If you regenerate code for the endpoint metadata definition project, the Informatica Connector Toolkit does not regenerate code for the user-exposed source code visible in the Informatica perspective. You have to manually edit the source code and make changes if you add, remove, or change the endpoint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

Configure advanced capabilities in swagger-based method

When you use the Swagger-based method to generate metadata for an object of hierarchical type, you can configure the pagination details for the API response.

You can use link-based or cursor-based pagination.

1. Click **Configure Advanced Capabilities**.



2. Select the pagination type.

- No Pagination. All the records are displayed in a single page.
- Link-based. The API response includes a field that contains the URL for the next page. For example, "next_page": "https://api.example.com/items?page=2"

The following image shows the sample response for link-based pagination:

```
493 |     {
494 |         "category": "forms",
495 |         "name": "Hotel Guest Registration Form",
496 |         "description": "Use this form to register hotel guests.",
497 |         "title": "Hotel Guest Registration Form",
498 |         "available": true,
499 |         "num_questions": 10,
500 |         "id": "2448",
501 |         "preview_link": "https://www.surveymonkey.com/z/Preview/?sm=A70_2Biq48JlhbXoZwFmPQIhqu7EISdfpdv6lCJzM6icAkXiV0pe1Go276y_2BwDztc"
502 |     },
503 | ],
504 | "per_page": 50,
505 | "page": 1,
506 | "total": 456,
507 | "links": {
508 |     "self": "https://api.surveymonkey.com/v3/survey_templates?per_page=50&page=1",
509 |     "next": "https://api.surveymonkey.com/v3/survey_templates?per_page=50&page=2",
510 |     "last": "https://api.surveymonkey.com/v3/survey_templates?per_page=50&page=10"
511 | }
512 | }
```


- Cursor-based. The API response includes a field that indicates whether there are more records available after the current set of records.

For example, "has_more": true

The following image shows the sample response for cursor-based pagination:

```

    "liveinstore": false,
    "marketing_features": [],
    "metadata": {
      "dryclean": "true",
      "unisex": "true"
    },
    "name": "Jacket",
    "package_dimensions": null,
    "shippable": null,
    "statement_descriptor": null,
    "tax_code": null,
    "type": "service",
    "unit_label": "1",
    "updated": 1644849050,
    "url": null
  },
  "has_more": true,
  "url": "/vi/products"
}

```

3. To use **Link-based** pagination type, configure the following fields:

- a. In the **Record Limit** field, enter the number of records to return for each page.

The screenshot shows a dialog box titled "Phase 2: Define Metadata > Configure Advanced Capabilities". The "Pagination" section is active, with the instruction "Enter the pagination details for the API response." The configuration fields are as follows:

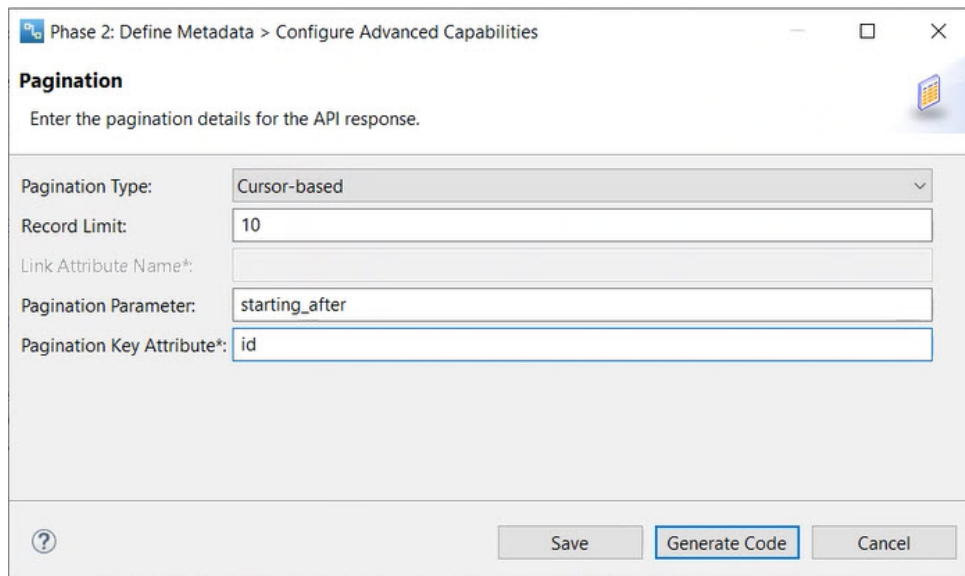
- Pagination Type:** Link-based (dropdown menu)
- Record Limit:** 10 (text input field)
- Link Attribute Name*:** next (text input field)
- Pagination Parameter:** (empty text input field)
- Pagination Key Attribute*:** (empty text input field)

At the bottom of the dialog, there are three buttons: a help icon (?), "Save", "Generate Code" (highlighted with a blue border), and "Cancel".

- b. In the **Link Attribute Name** field, enter the name of the field that contains the URL for the next page.

To use **Cursor-based** pagination type, configure the following fields:

- a. In the **Record Limit** field, enter the number of records to return per page.



Phase 2: Define Metadata > Configure Advanced Capabilities

Pagination

Enter the pagination details for the API response.

Pagination Type:

Record Limit:

Link Attribute Name*:

Pagination Parameter:

Pagination Key Attribute*:

- b. In the **Pagination Parameter** field, enter the name of the field that marks the starting point to retrieve the next set of records.
 - c. In the **Pagination Key Attribute** field, enter the name of the field that uniquely identifies each record.
4. Click **Generate code**.

Test the metadata

Specify the JVM parameters and connection details to test and debug the 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.

1. Click **Test Object Metadata**.

- Specify the JVM parameters for the virtual machine to debug the connector.

Phase 1: Establish Connection > Test

Test Settings

Specify the JVM parameters for the virtual machine to debug the connector.

Virtual Machine Parameters:

- Click **Next**.
- Enter the connection details.

Phase 2: Define Metadata > Test Object Metadata

Connection Details

Specify the connection details.

username * :

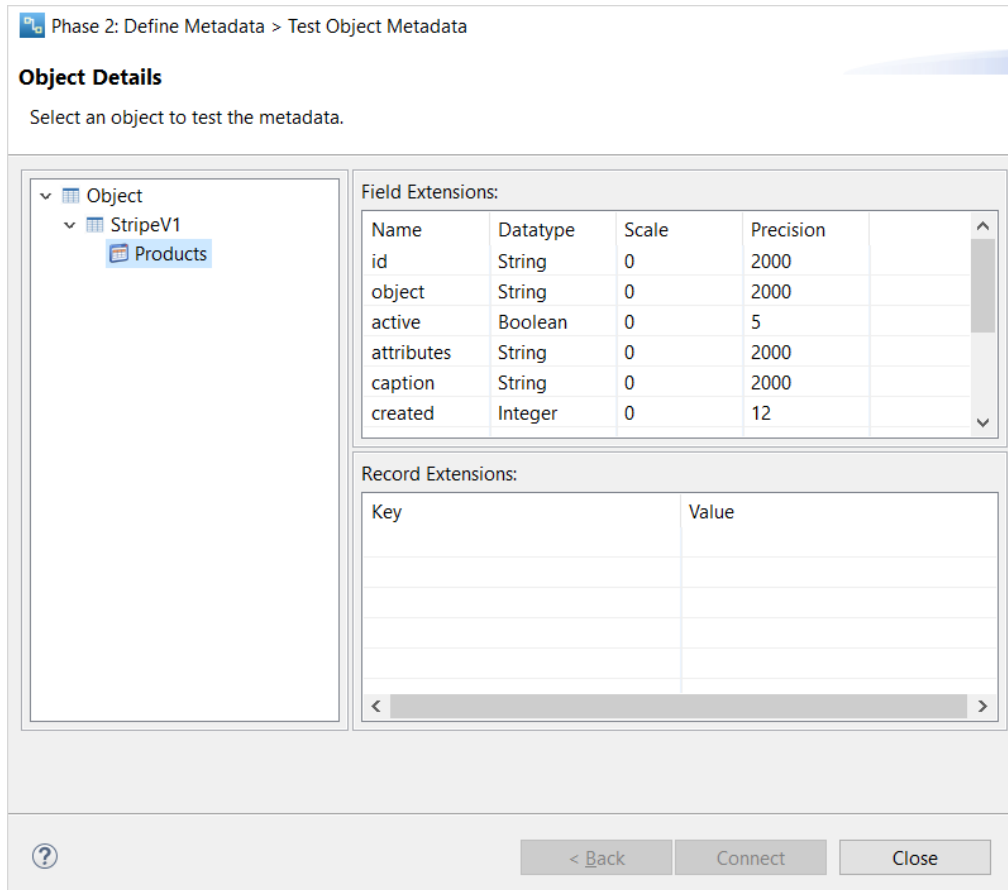
password * :

hostname * :

port * :

database * :

- Click **Connect**.
- Select an object to test the metadata.



7. Click **Connect**.

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
```

Phase 3: Configure the runtime 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 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 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.

Set up runtime

Configure the runtime behavior of the connector in Java. You can set up the runtime behavior for each object.

1. In the **Connector Progress view**, under **Phase 3: Configure the Runtime Behavior**, click **Set Up**.
The `<ConnectorID><NMOName>DataAdapter.java` and `<ConnectorID>DataConnection.java` files are generated.
2. Implement the methods in the `<ConnectorID><NMOName>DataAdapter.java` file based on the connector type.
 - a. Implement the following methods for all connector types:
 - `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`
 - `reset` (Optional. Implement this method if the connector supports the lookup operation.)
 - `beginDataSession`
 - `endDataSession`
 - b. Implement the following methods for a JDBC driver-based connector:
 - `read`
 - `write`
 - c. Implement the following methods for a REST API-based connector:
 - `call`
 - `executeRestAPI`

The auto-generated code for the call and executeRestAPI methods contains the connection attributes for API failover. You can edit the attribute values according to your requirement.

- `ConnectionTimeout`. The wait time in milliseconds to get a response from a REST endpoint. The connection attempt times out if it does not receive a response within the specified timeout period. Default is 120000 milliseconds.
 - `connectiondelaytime`. The delay time in milliseconds to wait after a request to a REST endpoint fails and before it attempts to resend requests to the REST endpoint. Default is 10000 milliseconds.
 - `retryattempts`. Number of times a connection can attempt to reconnect when 400 and 500 series error codes are returned in the response. Default is 3. Specify 0 to disable the retry attempts.
3. Implement the following methods in the `<ConnectorID>DataConnection.java` file:
 - `connect`
 - `disconnect`
 4. To support data preview in the Informatica Intelligent Cloud Services, implement the following method in the `<ConnectorID>ASOOperationObjMgr.java` file: `prepareRuntimeOperation`

Test runtime behavior

Test the runtime behavior of the connector to test the connection definition, metadata of the data source, and operations that the connector supports. Use the same debug configuration that you used to test the connection and metadata components of the connector.

You can test both the read and write capabilities of a connector.

Test the read capability

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. Under **Test Read Capability**, click **Test Runtime Behavior**.

The default JVM environment settings and tracing level are displayed. 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

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. Under **Test Write Capability**, click **Test Runtime Behavior**.

The default JVM environment settings and tracing level are displayed. 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**.

Create messages

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 handling 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.createNlsAdapterSDKException(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.

Phase 4: Generate the connector package

Specify the connector details and generate the connector package. The Informatica Connector Toolkit bundles connector artifacts and generates the connector package that you can use for the selected cloud service.

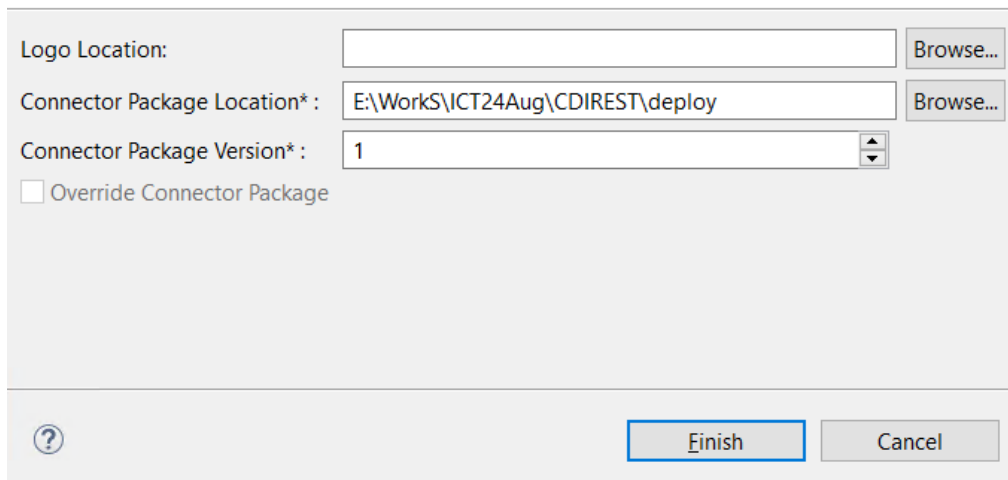
1. In the **Connector Progress** view, under **Phase 4: Generate the Connector Package**, click **Generate Package**.

The Generate Connector Package window appears.

Phase 4: Generate Connector Package

Connector Package Settings

Specify the settings to generate the connector package.




Logo Location: Browse...

Connector Package Location* : E:\WorkS\ICT24Aug\CDIREST\deploy Browse...

Connector Package Version* : 1

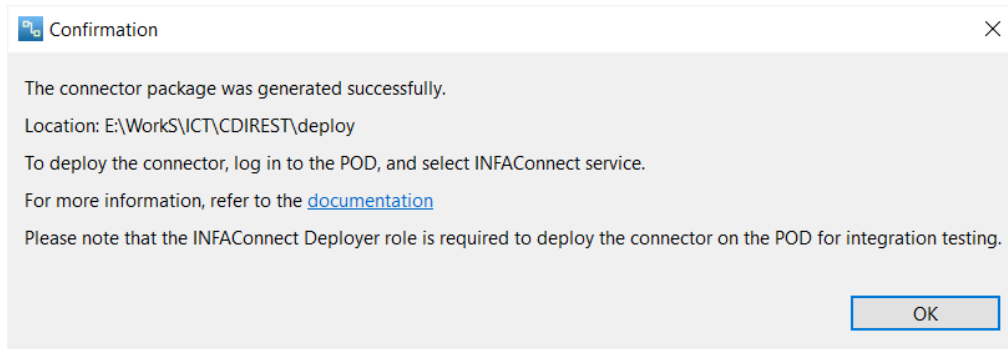
Override Connector Package

 Finish Cancel

2. Specify the connector details to generate the connector package.

Property	Description
Logo Location	The image file for the logo of the connector. The file must be in PNG or JPEG format and the file size must not exceed 64 KB. The dimensions of the logo must be 89 x 45 pixels. The file name must not contain spaces or special characters.
Connector Package Location	The location where you want to save the connector package.
Connector Package Version	The version of the connector package.
Override Connector Package	Select to override the existing connector package.

3. Click **Finish**.
If the connector package is generated successfully, a confirmation message is displayed.



4. Click **OK**.

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

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

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

Phase 5: Generate and run the test scenarios

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.

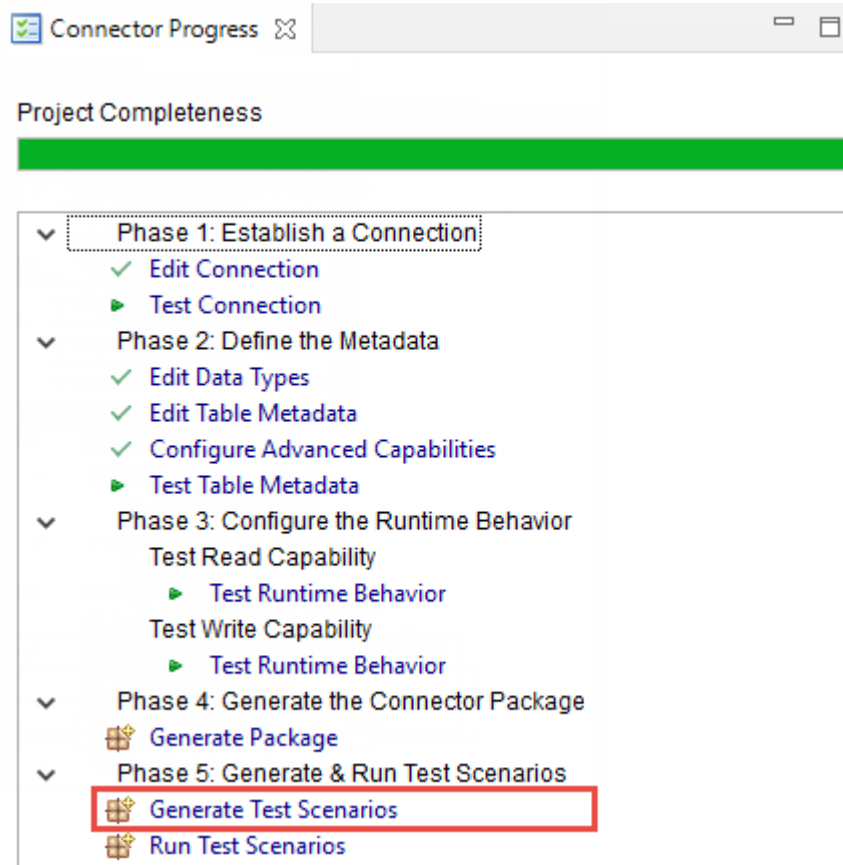
When the automation framework runs, it performs the following tasks:

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.

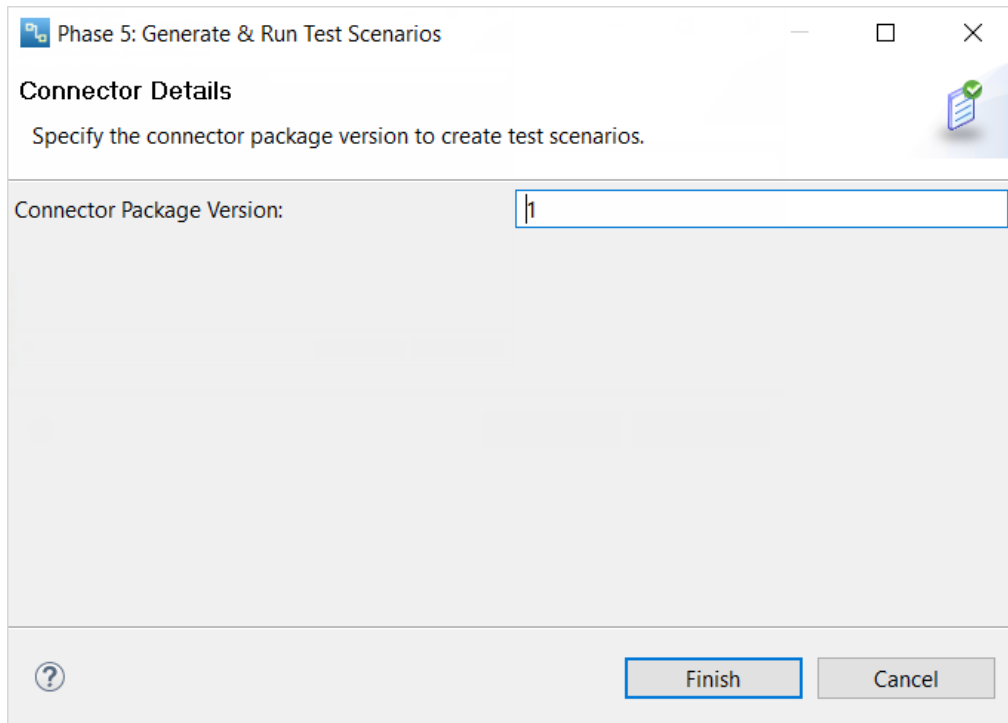
Generate the test scenarios

Specify the connector package version and generate test scenarios to perform unit and integration testing for your connector. The generated test scenarios are available in the Test Scenarios folder on the Connector Navigator tab.

1. In the **Connector Progress** view, under **Phase 5: Generate & Run Test Scenarios**, click **Generate Test Scenarios**.



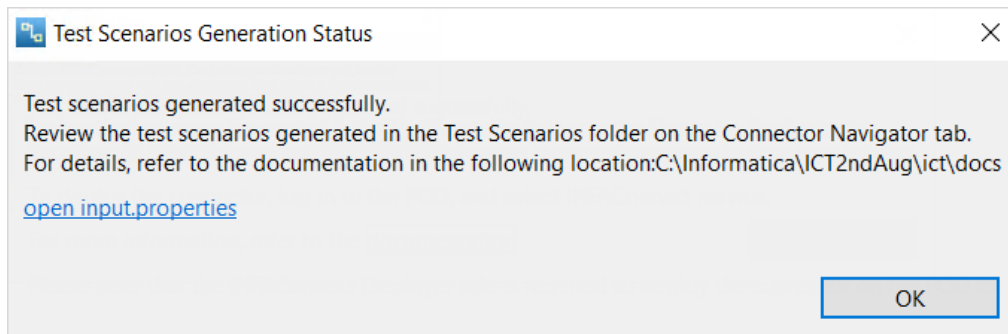
2. Specify the connector package version.



Ensure that you specify the same version number that you provided in Phase 4 before you generate the connector package.

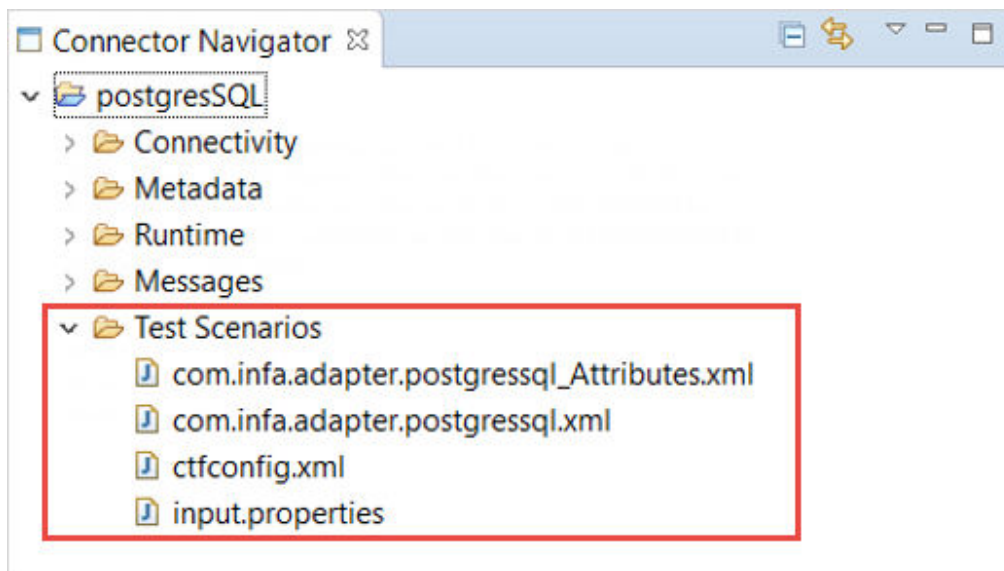
3. Click **Finish**

A confirmation message appears.



4. Click **OK**.
5. On the **Connector Navigator** tab, navigate to the **Test Scenarios** folder for the connector. The following files are generated in the Test Scenarios folder:
 - Attributes file
 - TestNG suite file
 - ctfconfig.xml file

- input.properties file



The following table describes the files generated in the test scenarios folder:

Files	Description
Attribute File	Generated as <code>com.infa.adapter.<adapter_name>_Attributes.xml</code> in the Test Scenarios folder. It consists of capability attributes such as read, write, and connection attributes. Specify the values for these properties.
TestNG Suite File	Generated as <code>com.infa.adapter.<adapter_name>.xml</code> in the Test Scenarios 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 <code>testsuite</code> file to add the actual values for the parameters. - Specify the values in the <code>input.properties</code> file.
<code>ctconfig.xml</code>	File that contains the required entries to run the test cases.
<code>input.properties</code>	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.

6. Open the TestNG Suite file.

A new tag named SANITY is added and the CAT and PAT tags are renamed to UNIT and INTEGRATION respectively.

```
--><test name="test_connection">
  <parameter name="tags" value="UNIT, INTEGRATION, SANITY"/>
  <parameter name="ConnectionName" value="connection1"/>
  <classes>
    <class name="com.informatica.cci.direct.CCITestConnection"/>
  </classes>
</test>
```

Configure parameters in 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.

```
61
62<test name="basic_read">
63  <parameter name="tags" value="CAT"/>
64  <parameter name="ConnectionName" value="connection1"/>
65  <parameter name="ReaderProperty" value="read1"/>
66  <parameter name="objectName1" value="$_EP{READ.objectName1}"/>
67  <classes>
68    <class name="com.informatica.cci.direct.CCIRuntimeReaderAllExpr"/>
69  </classes>
70 </test>
71
```

You can perform one of the following tasks:

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

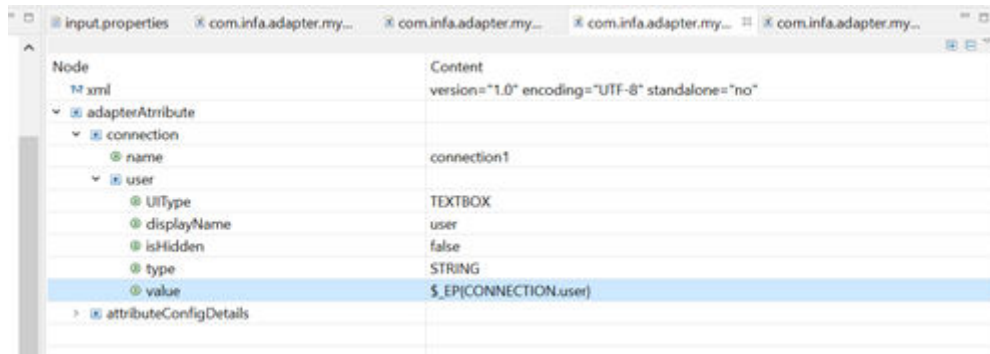
```
61
62<test name="basic_read">
63  <parameter name="tags" value="CAT"/>
64  <parameter name="ConnectionName" value="connection1"/>
65  <parameter name="ReaderProperty" value="READ"/>
66  <parameter name="objectName1" value="public/employee"/>
67  <classes>
68    <class name="com.informatica.cci.direct.CCIRuntimeReaderAllExpr"/>
69  </classes>
70 </test>
```

- 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:

```
21##### Source Object for Reader testcases #####
22READ.objectName1=public/employee
23
```

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.

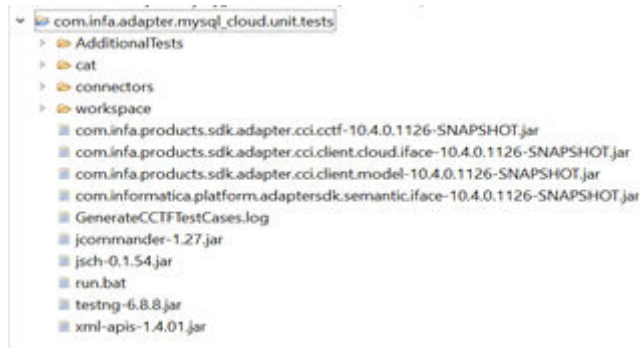


Regenerate a test case

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** view, 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.

Run the test scenarios

Run the test scenarios to validate the unit and integration tests and populate a unified test report with the test results. View the results in eclipse or browser. See the `ctfrun.log` file in the result folder for a detailed log. The result folder is available in the following path: `<Path to the connector folder>/cctf/cat/result`.

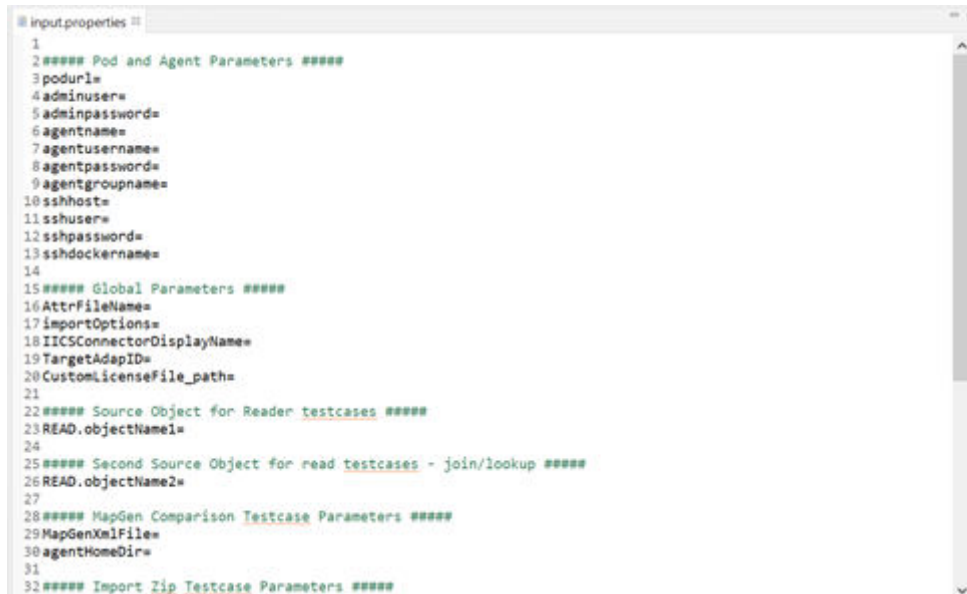
You can then view the success or failure status in the test report.

Unit tests and sanity tests

Perform the following steps to run the unit tests and sanity tests:

1. In the **Connector Navigator** view, navigate to the **Test Scenarios** folder for the connector.
2. Open the `input.properties` file and specify the parameters to execute the test scenarios.

The file contains a list of 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.



```
1
2#### Pod and Agent Parameters ####
3podurl=
4adminuser=
5adminpassword=
6agentname=
7agentusername=
8agentpassword=
9agentgroupname=
10sshhost=
11sshuser=
12sshpassword=
13sshdockername=
14
15#### Global Parameters ####
16AttrFileName=
17importOptions=
18IICSConnectorDisplayName=
19TargetAdapID=
20CustomLicenseFile_path=
21
22#### Source Object for Reader testcases ####
23READ.objectName1=
24
25#### Second Source Object for read testcases - join/lookup ####
26READ.objectName2=
27
28#### MapGen Comparison Testcase Parameters ####
29MapGenXmlFile=
30agentHomeDir=
31
32#### Import Zip Testcase Parameters ####
```

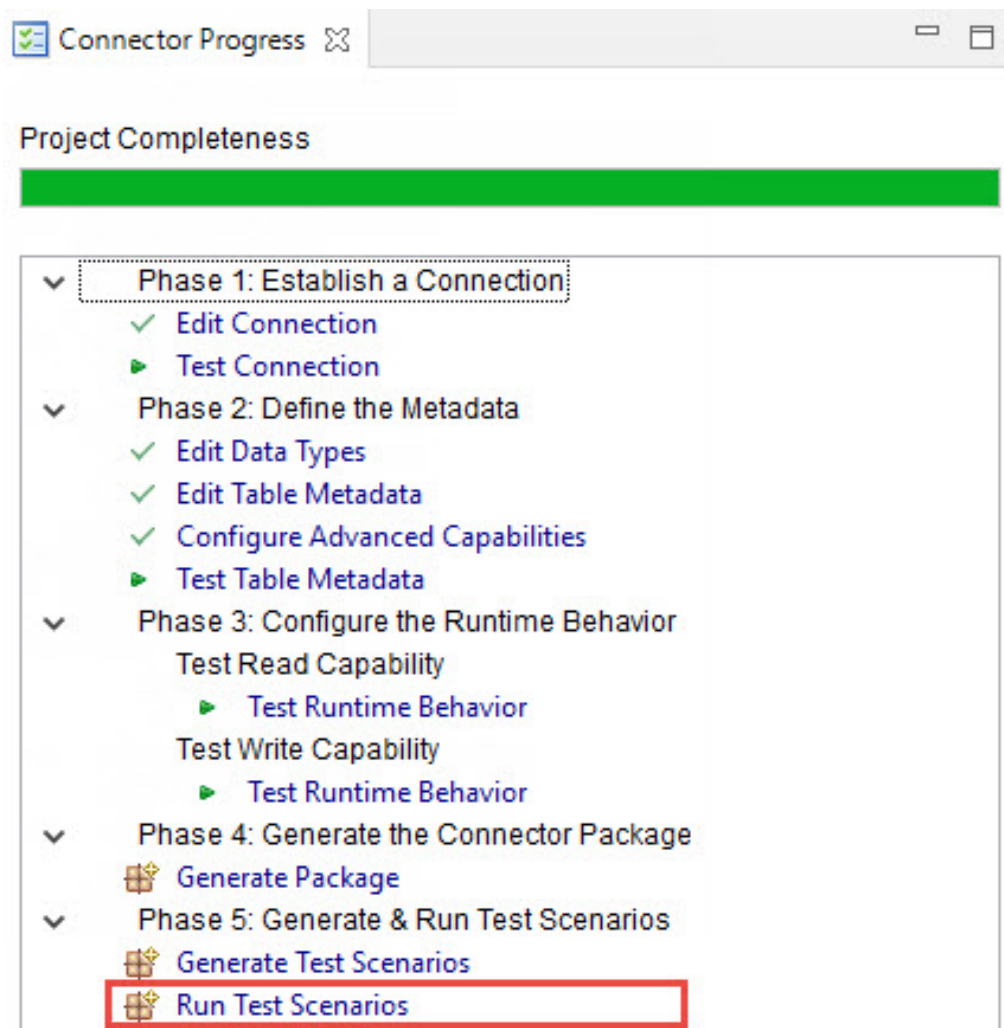
3. Edit the tag values in the `testNG` suite file. Specify CAT for Code Acceptance Test or Unit Test and PAT for Product Acceptance Test or Integration Test.



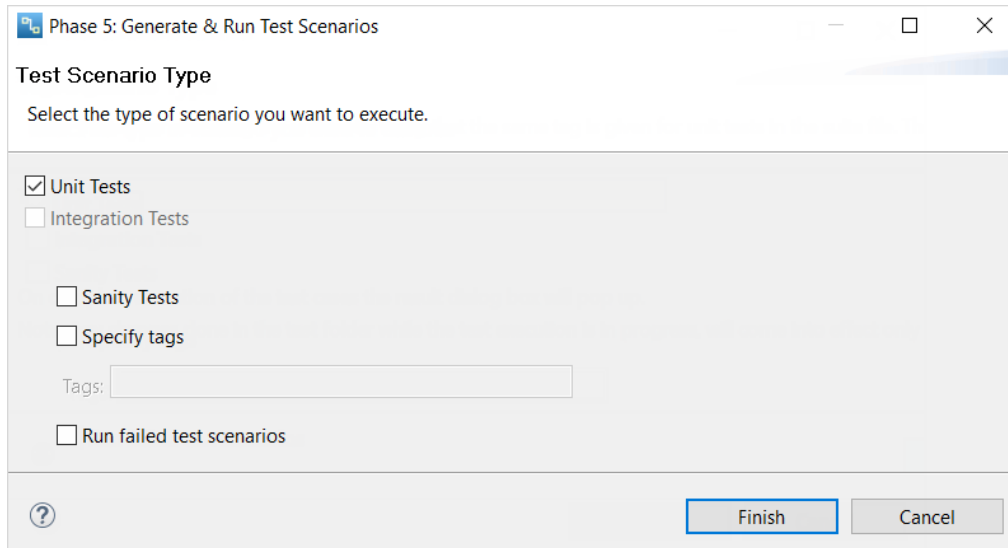
```
27<!--><test name="test_connection">
28  <parameter name="tags" value="CAT,PAT"/>
29  <parameter name="ConnectionName" value="connection1"/>
30  <classes>
31    <class name="com.informatica.cci.direct.CCITestConnection"/>
32  </classes>
33 </test>
34
35<test name="record_information">
36  <parameter name="tags" value="CAT"/>
37  <parameter name="ConnectionName" value="connection1"/>
38  <parameter name="objectName1" value="$_EP{READ.objectName1}"/>
39  <classes>
40    <class name="com.informatica.cci.direct.CCIMetadataRecordInformation"/>
41  </classes>
42 </test>
43
44<test name="record_field_detail">
45  <parameter name="tags" value="PAT"/>
46  <parameter name="ConnectionName" value="connection1"/>
47  <parameter name="objectName1" value="$_EP{READ.objectName1}"/>
48  <classes>
49    <class name="com.informatica.cci.direct.CCIMetadataRecordDetail"/>
50  </classes>
51 </test>
52
53<test name="record_search">
54  <parameter name="tags" value="NONE"/>
55  <parameter name="ConnectionName" value="connection1"/>
56  <parameter name="objectName1" value="$_EP{READ.objectName1}"/>
57  <classes>
58    <class name="com.informatica.cci.direct.CCIMetadataSearch"/>
59  </classes>
60 </test>
61
```

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

4. In the **Connector Progress** view, under **Phase 5: Generate & Run Test Scenarios**, click **Run Test Scenarios**.

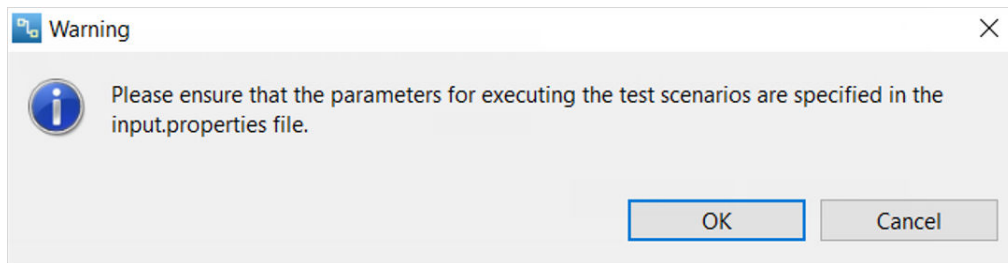


5. Select **Units Tests** and **Sanity Tests**.

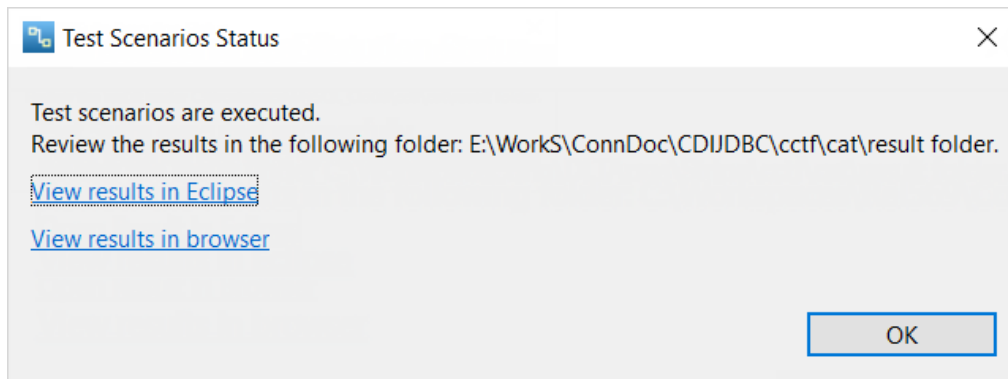


- Specify the tags to run specific unit tests.
- When you configure the **Units Tests** or **Sanity Tests** for the first time, the **Run failed test scenarios** option is not enabled.
After the test scenarios are executed, you can select the **Run failed test scenarios** option to execute only the failed test scenarios.
- Click **Finish**.

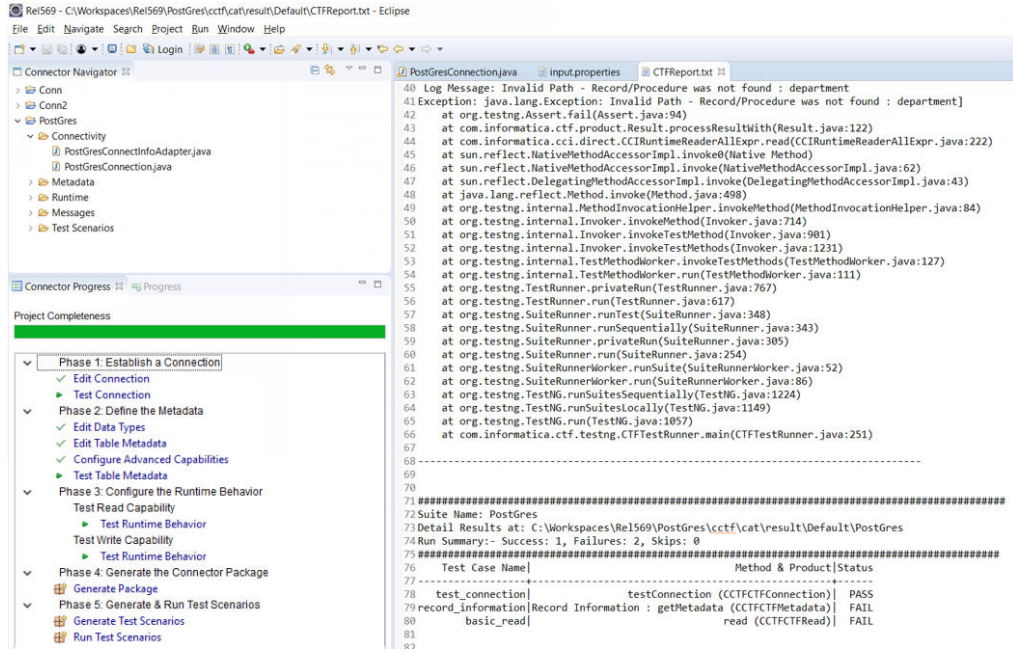
A warning window appears.



- Click **OK**.
After the test scenarios are executed, the **Test Scenarios Execution Status** window appears.



- Click **View results in Eclipse** to view the results in Eclipse.



Optionally, click **View results in browser** to view the results in the browser.

Test	# Passed	# Skipped	# Failed	Time (ms)	Included Groups	Excluded Groups
PostGres						
test_connection	1	0	0	5,831	SANITY	
record_information	0	0	1	200	SANITY	
record_field_detail	0	0	0	0	SANITY	
record_search	0	0	0	0	SANITY	
record_search_recordFilterType	0	0	0	0	SANITY	
record_search_recordFilterName	0	0	0	0	SANITY	
record_search_recordFilterDescription	0	0	0	0	SANITY	
record_search_recordFilterLabel	0	0	0	0	SANITY	
record_search_allFilter	0	0	0	0	SANITY	
basic_read	0	0	1	156	SANITY	
read_customBaselineTgt_filePath	0	0	0	0	SANITY	
read_portsTo_connect	0	0	0	0	SANITY	
mapGenXml_comparison	0	0	0	0	SANITY	
sessionLog_comparison	0	0	0	0	SANITY	
verifyMessage_for_testComparison	0	0	0	0	SANITY	
negative_testcase	0	0	0	0	SANITY	
dataVerification_skipNull	0	0	0	0	SANITY	
importZip_executeMCT	0	0	0	0	SANITY	
Total	1	0	2	6,187		

Class	Method	Start	Time (ms)
PostGres			
test_connection — passed			

11. Click **OK**.
12. To view the detailed log, see the `ctfrun.log` file in the result folder.
The folder also contains the `testing` and `junit` reports.

Integration tests

Perform the following steps to run the integration tests:

1. On the **Connector Navigator** tab, navigate to the **Test Scenarios** folder for the connector and open the `ctfconfig.xml`.

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.

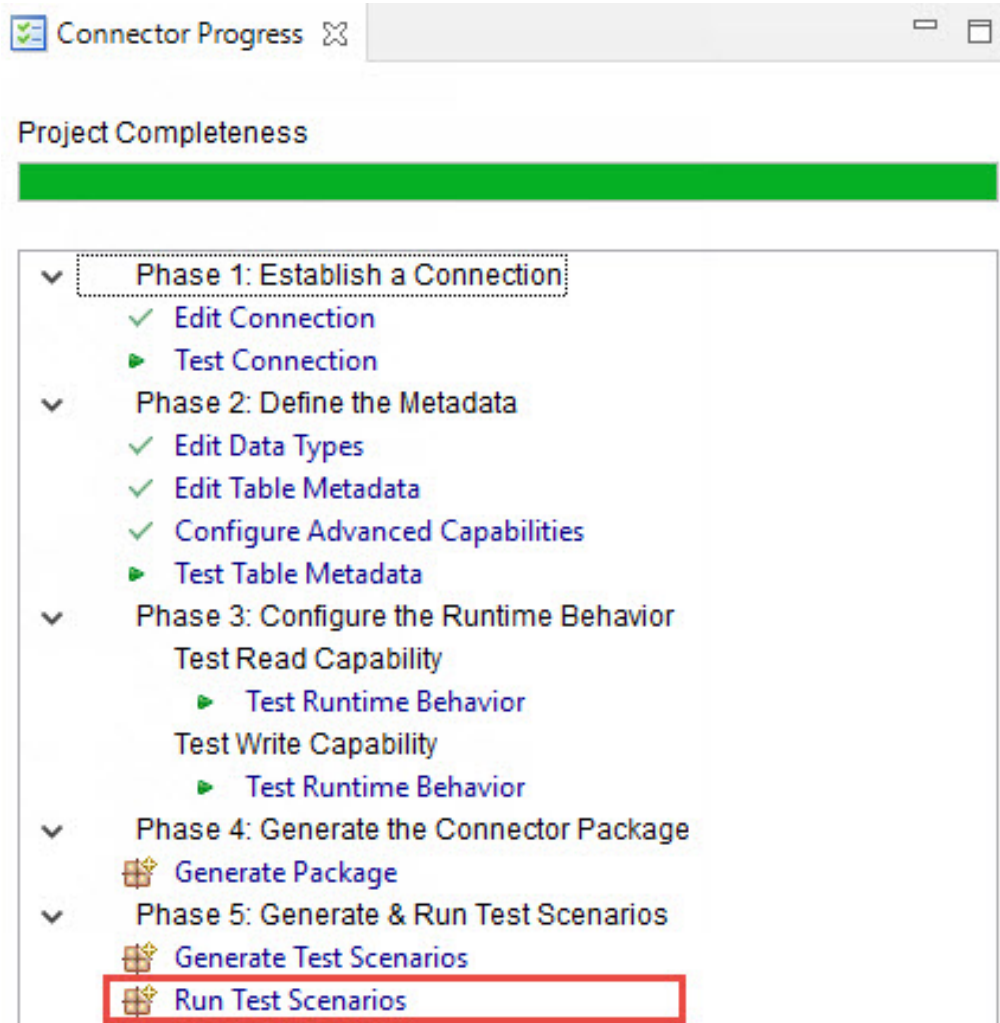
2. Specify the pod details in the `ctfconfig.xml` file or in the `input.properties` file.

Node	Content
xml	version="1.0" encoding="UTF-8" standalone="yes"
CTF	
Products	
CloudSecureAgent	
podURL	\$_EP(podurl)
adminUser	\$_EP(adminuser)
adminPassword	\$_EP(adminpassword)
agent	
name	\$_EP(agentname)
user	\$_EP(agentusername)
password	\$_EP(agentpassword)
groupName	\$_EP(agentgroupname)
sshHost	\$_EP(sshhost)
sshUser	\$_EP(sshuser)
sshPassword	\$_EP(sshpassword)
dockerName	\$_EP(sshdockername)
CCTF	
TestNGSuites	
resultLocation	result
Suite	
name	
file	com.infa.adapter.mysql_cloud.xml

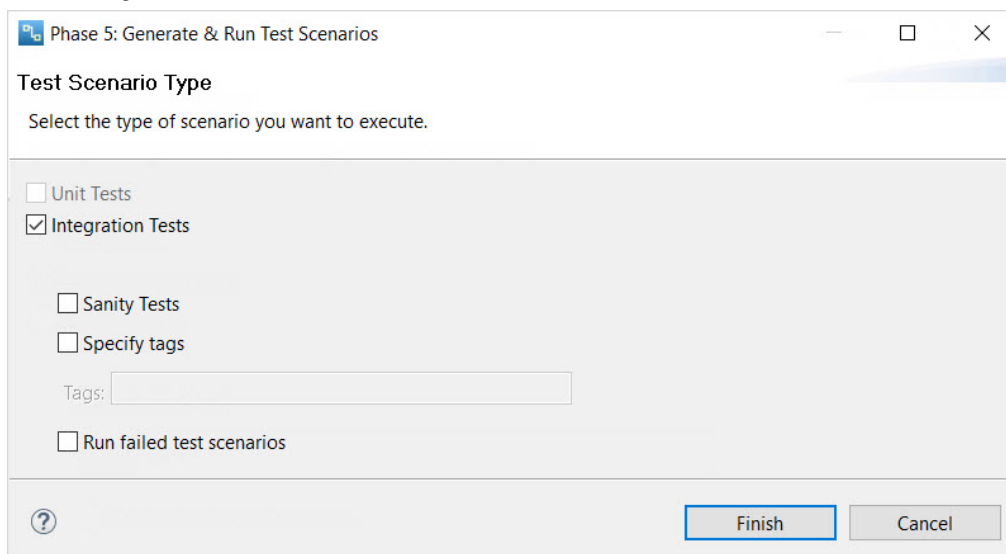
The following table describes the keys in the `ctfconfig.xml` file:

Key	Description
podurl	Pod URL for log in to the pod.
adminuser	Administrator user for the pod. Specify this parameter 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.
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 file.
Specify CAT for Code Acceptance Test or Unit Test and PAT for Product Acceptance Test or Integration Test.
Set the value of the tags to `NONE` during the test case review to skip the test case execution.
4. In the **Connector Progress** view, under **Phase 5: Generate & Run Test Scenarios**, click **Run Test Scenarios..**

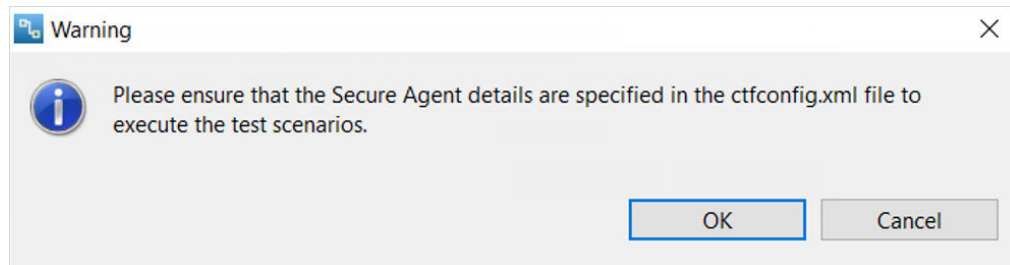


5. Select **Integration Tests**.

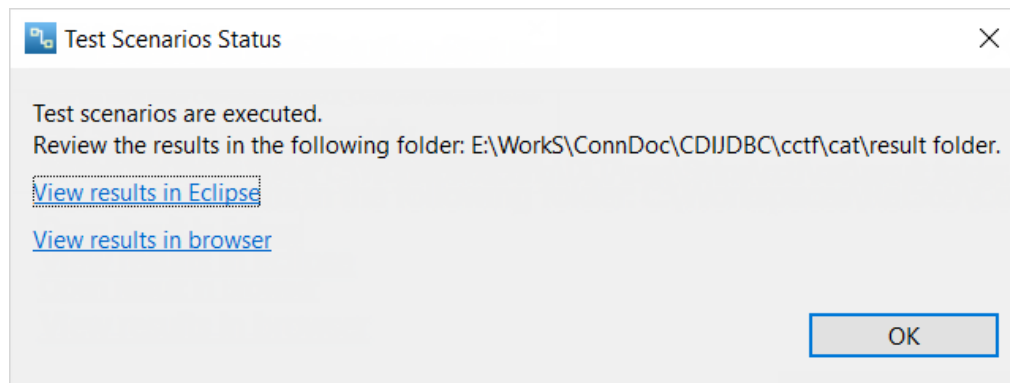


6. Specify the tags to run specific integration tests.

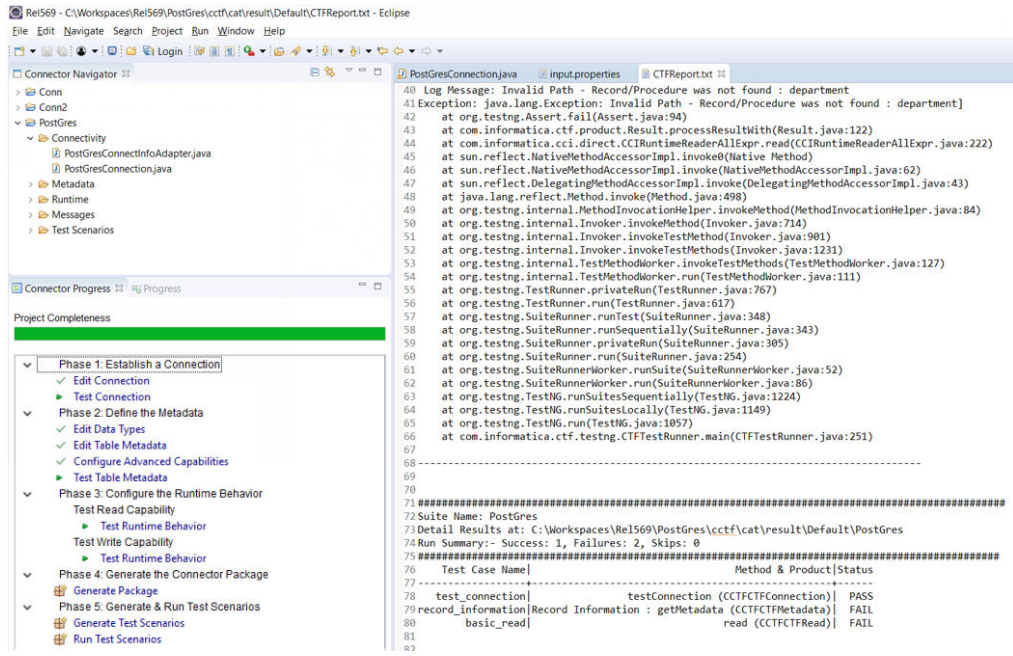
7. When you configure the **Integration Tests** for the first time, the **Run failed test scenarios** option is not enabled.
After the test scenarios are executed, you can select the **Run failed test scenarios** option to execute only the failed test scenarios.
8. Click **Finish**.
A warning window appears.



9. Click **OK**.
After the test scenarios are executed, the **Test Scenarios Execution Status** window appears.



10. Click **View results in Eclipse** to view the results in Eclipse.



Optionally, click **View results in browser** to view the results in the browser.

Test	# Passed	# Skipped	# Failed	Time (ms)	Included Groups	Excluded Groups
PostGres						
test_connection	1	0	0	5,831	SANITY	
record_information	0	0	1	200	SANITY	
record_field_detail	0	0	0	0	SANITY	
record_search	0	0	0	0	SANITY	
record_search_recordFilterType	0	0	0	0	SANITY	
record_search_recordFilterName	0	0	0	0	SANITY	
record_search_recordFilterDescription	0	0	0	0	SANITY	
record_search_recordFilterLabel	0	0	0	0	SANITY	
record_search_allFilter	0	0	0	0	SANITY	
basic_read	0	0	1	156	SANITY	
read_customBaselineTgt_filePath	0	0	0	0	SANITY	
read_portsTo_connect	0	0	0	0	SANITY	
mapGenXml_comparison	0	0	0	0	SANITY	
sessionLog_comparison	0	0	0	0	SANITY	
verifyMessage_for_testComparison	0	0	0	0	SANITY	
negative_testcase	0	0	0	0	SANITY	
dataVerification_skipNull	0	0	0	0	SANITY	
importZip_executeMCT	0	0	0	0	SANITY	
Total	1	0	2	6,187		

Class	Method	Start	Time (ms)
PostGres			
test_connection — passed			

11. Click **OK**.
12. To view the detailed log, see the `ctfrun.log` file in the result folder.
The folder also contains the `testng` and `junit` reports.

CHAPTER 4

Develop 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:

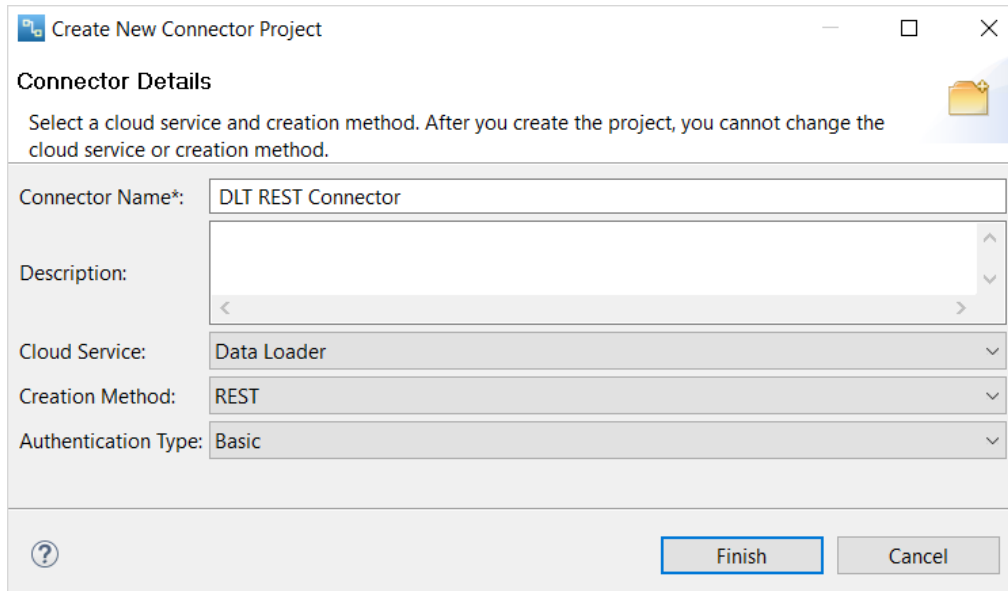
1. Create a connector project and select the product type. Then, select the creation method and define the basic properties of the connector such as name, ID, and vendor name.
2. Define the connection attributes to connect to the data source. Implement the methods to open or close the connection to the 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.

Create a new connector project

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. On the Eclipse IDE toolbar, click the **Create New Connector** button (). The **Create New Connector Project** window appears.

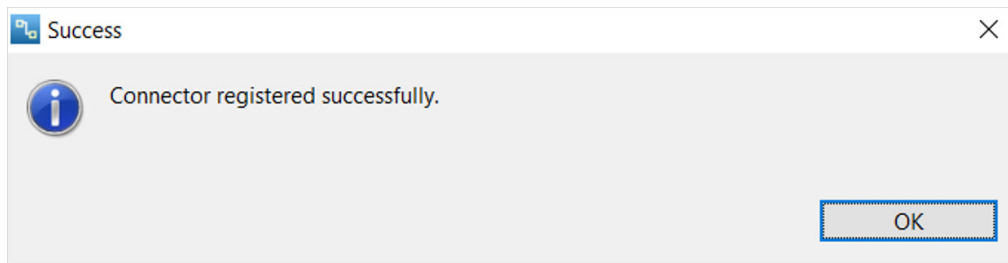


2. In the **Create New Connector Project** window, configure the following properties:

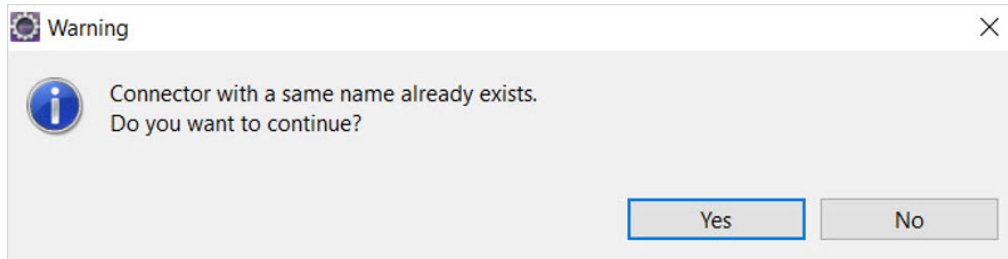
Property	Description
Connector Name	Name of the connector. Specify a unique name for the connector. The connector name must be an alphanumeric string. The first character of the name must be a letter.
Description	Description of the connector.
Cloud Service	The cloud service for which you want to create a connector. Select Data Loader .

Property	Description
Creation Method	<p>The method to create a connector.</p> <p>Select one of the following creation methods:</p> <ul style="list-style-type: none"> - JDBC. Create a JDBC driver-based connector. - Libraries. Create a connector using a third-party library SDK. - REST. Create a connector for REST-based application. <p>Default is REST.</p>
Authentication Type	<p>The authentication type for the REST creation method.</p> <p>Select one of the following authentication types:</p> <ul style="list-style-type: none"> - No Authentication: Does not require authentication. - Basic. Uses the 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. <p>Default is basic authentication.</p>

3. If a connector with the same name already exists, the following error is displayed:
A connector with the same name already exists. Enter a different name.
Click **OK** and specify a unique name for the new connector.
4. Click **Finish**.
If the new connector is successfully registered, a success message appears.



5. Click **OK**.
The new connector project appears on the **Connector Navigator** tab.
6. If the connector is already registered, a warning appears. Click **Yes** to skip the registration process.



Phase 1: Establish a connection

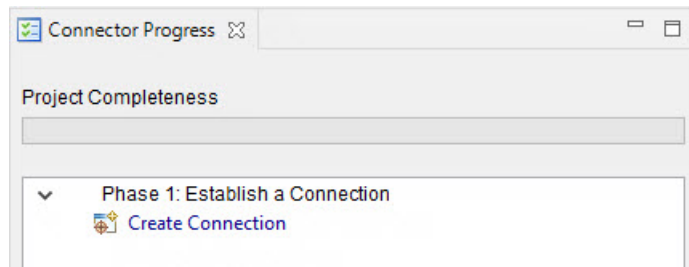
The connection attributes of a connector determine how the connector connects to the data source.

Use the Informatica Connector Toolkit to define the connection attributes, create connection pages, and add third-party libraries required to connect to the data source.

Create a connection

Define the connection attributes, create connection pages, and add third-party libraries required to create a connection.

1. In the **Connector Progress** view, under **Phase 1: Establish a Connection**, click **Create Connection**.



2. Configure the connection attributes.

Phase 1: Establish Connection > Create

Connection Attributes

Use the default connection attributes or add, edit, or delete attributes of your choice.

Name	Display Na...	Data Type	Mandatory	Encrypted	Has Depen...
baseURL	Base URL	String	✓	✗	✗
username	username	String	✓	✗	✗
password	password	String	✓	✓	✗

Buttons: Add, Edit, Delete

Navigation: ? < Back, Next >, Save, Generate Code, Cancel

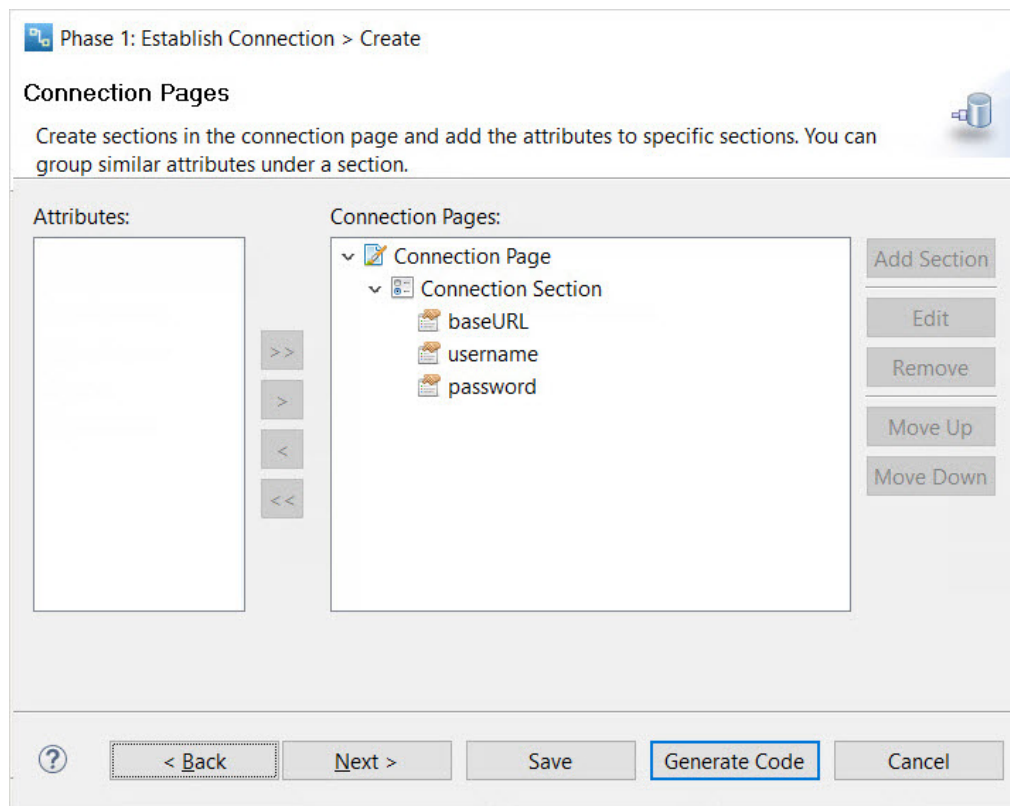
- a. Click **Add** to add a new connection attribute.
The attributes that you configure correspond to the connection object properties.

- b. In the **Add Attribute** window, enter the following properties for the connection attribute:

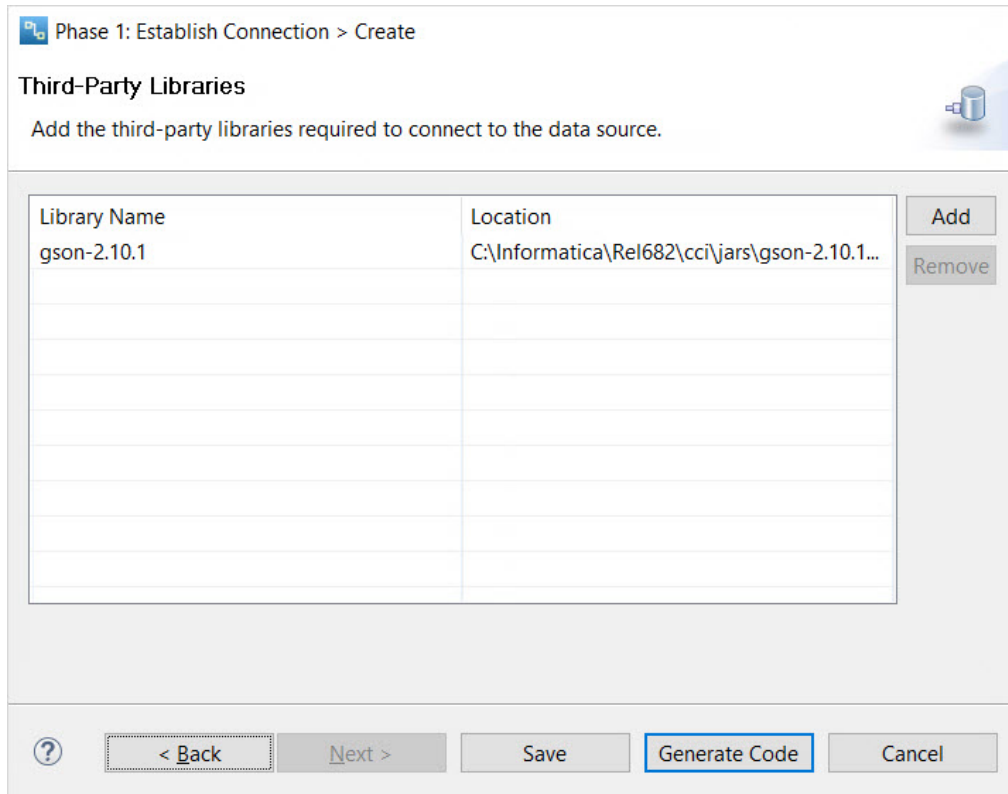
Attribute Property	Description
Name	Name of the connection attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the attribute.
Default Value	Default value for the attribute.
Min Length	Minimum length for the value of the attribute.
Max Length	Maximum length for the value of the attribute.
Encrypted	Indicates whether you can encrypt the attribute.
Supports Parameter	Indicates whether you can parameterize 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.
Has Dependent Fields	Indicates whether the attribute has dependent fields.
Allowed Values	List of values allowed for the attribute.

- c. Click **OK**.
3. Click **Next**.
4. Configure the connection pages.



- a. To create a new connection section, click **Add Section** and enter the section name, section title, tool tip, and section description.
- b. Add attributes to specific sections. You can group similar attributes under a section.
- c. To change the order in which connection sections appear in a connection page, use **Move Up** or **Move Down**.
5. Click **Next**.
6. Add the third-party libraries.
Click **Add** to add the third-party libraries required to connect to the data source.



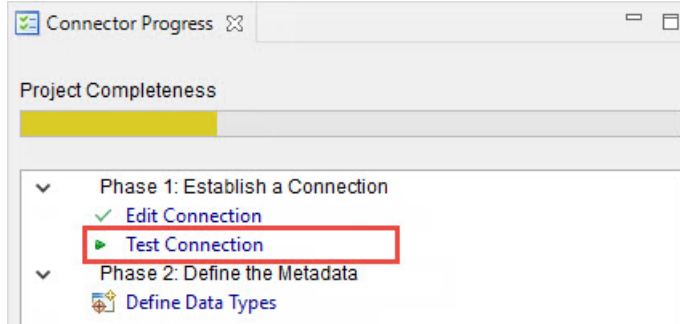
7. Click **Generate Code**.
After you define the connection attributes, the Informatica Connector Toolkit generates the following Java files:
 - <ConnectorName>ConnectInfoAdapter.java
 - <ConnectorName>Connection.java
8. Update the `ConnectInfoAdapter.java` file to implement connection validation and attribute dependencies. Also, update the `Connection.java` file to implement the methods that open and close connection to the data source.
9. 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.

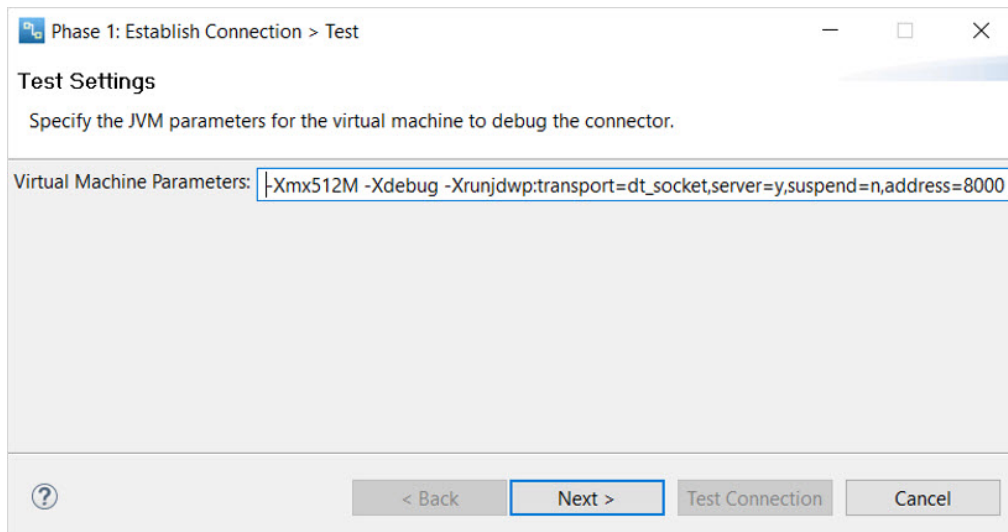
Test a connection

Specify the JVM parameters and connection details to test and debug the connection to the data source.

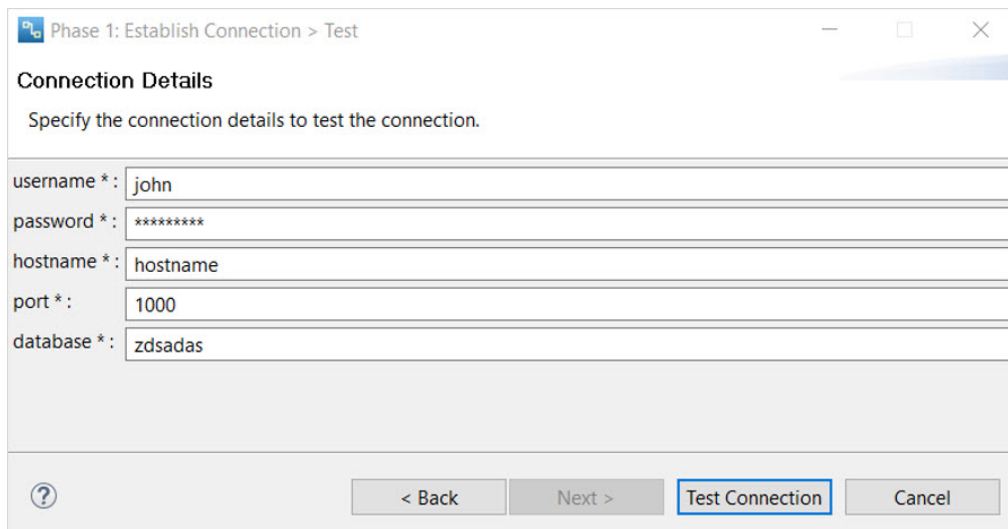
1. Click **Test Connection**.



2. Specify the JVM parameters for the virtual machine to debug the connector.



3. Click **Next**.
4. Specify the connection details to test the connection.



5. Click **Test Connection**.

Phase 2: Define the metadata

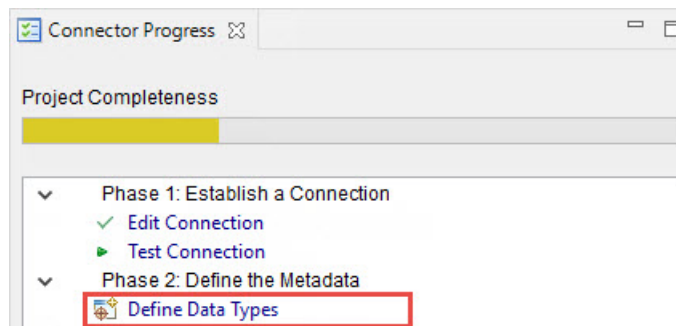
The connector metadata represents the metadata in the data source for which you build the connector.

The first time you add object metadata for a data loader connector, you can configure only the read capability. You must edit the object metadata to add additional metadata information for the objects and fields.

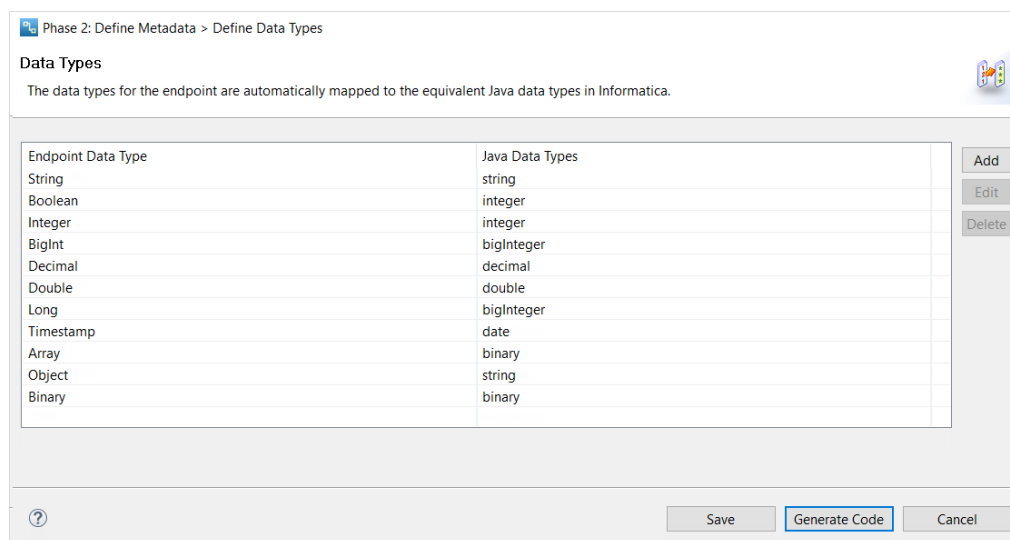
Define the data types

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.

1. In the **Connector Progress** view, under **Phase 2: Define Metadata**, click **Define Data Types**.



The data types for the endpoint are automatically mapped to the equivalent Java data types in Informatica.



2. Perform the following steps to add a data type:
 - a. Click **Add**.
 - b. In the **Add Data Type** window, configure the following properties for the data type:

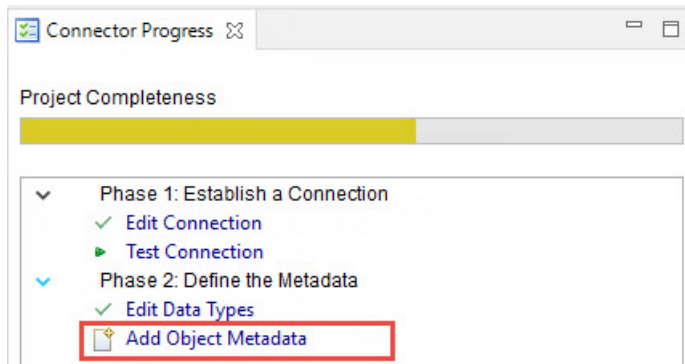
Property	Value
Type Name	Name for the data type.
Comments	Comments for the data type.
Best Java Type for Read	The Java data type in Informatica that best maps to the endpoint data type when the connector reads from the data source.
Properties	<p>Set one or more of the following properties for the data type:</p> <ul style="list-style-type: none"> - 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. <p>The maximum length of any attribute cannot be greater than Integer.MAX_VALUE, which is 2,147,483,647.</p> <ul style="list-style-type: none"> - Unit of length such as characters, bytes, and bits. - Date properties such as hour, minute, second, year, month, day, and time zone.

- c. Click **OK**.
3. Click **Generate Code**.

Endpoint object metadata

Add object metadata for a connector for Data Loader.

1. Click **Add Object Metadata**.



2. Select **Enable Read Capability** to define read capability for the connector.

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.
Override Partitions	Indicates if the attribute can be overridden for each partition. Implement the <ConnectorID><NMOName>AutoPartitioningMetadataAdapter file to define the partition support.
Allowed Values	List of values allowed for the attribute.

b. Click **OK**.

3. Click **Generate Code**.

After you define the connector metadata, the Informatica Connector Toolkit generates the <NMOName>MetadataAdapter.java file in the **Metadata** folder.

4. 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 object metadata

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

1. Click **Edit** for the endpoint.
2. Enter the object details.

Property	Description
Name	Name for the object.
Display Name	Display name of the object.
Description	Description of the object.

3. Click **Next**.

4. To add additional metadata information for the object, select **Add Record Extension**.

Phase 2: Define Metadata > Add Object Metadata

Endpoint Object Definition

Enter the endpoint metadata record details.

Add Record Extension

Name	Display Na...	Description	Data Type	Mandatory	Encrypted	Default Val...

- a. Click **Add** to add a record extension and configure the following properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the 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.

Property	Description
Hidden	Determines whether you want to display or hide the attribute.
Allowed Values	List of values allowed for the attribute.

- b. Click **OK**.
5. Click **Next**.
 6. To add additional metadata information for the fields, select **Add Field Extension**.

- a. Click **Add** to add a field extension and configure the following properties:

Property	Description
Name	Name of the attribute.
Display Name	Display name for the attribute.
Description	Description of the attribute.
Data Type	Data type of the attribute.
Default Value	Default value for the attribute.
Min Length	Minimum length for the value of the 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.
Supports Parameter	Indicates whether you can parameterize the attribute. When you parameterize an attribute, you can assign values for the attribute at run time. <ul style="list-style-type: none"> - Full Parameterization. Indicates you can parameterize the value of an attribute completely. - Partial Parameterization. You can parameterize a part of the attribute value. - No. The attribute does not support parameterization.
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.

b. Click **OK**.

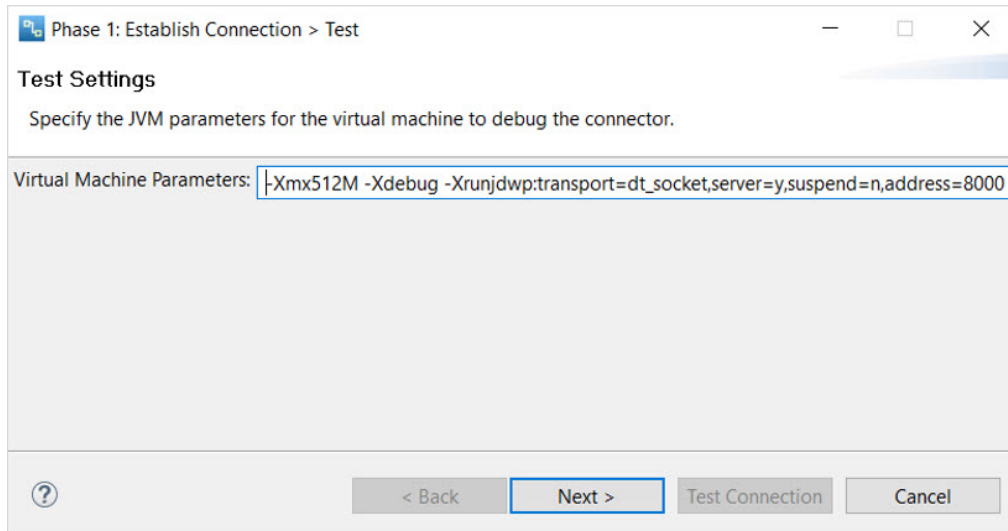
7. Click **Generate Code**.

Test the metadata

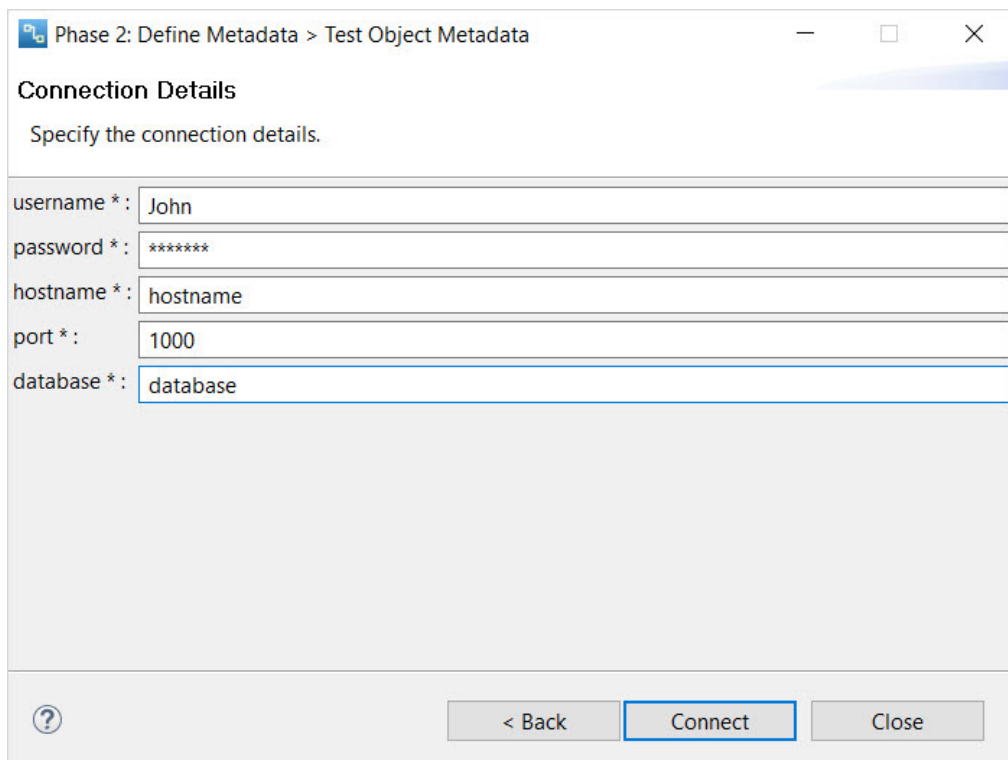
After you define the endpoint metadata objects, you can test metadata that you import from the data source.

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.

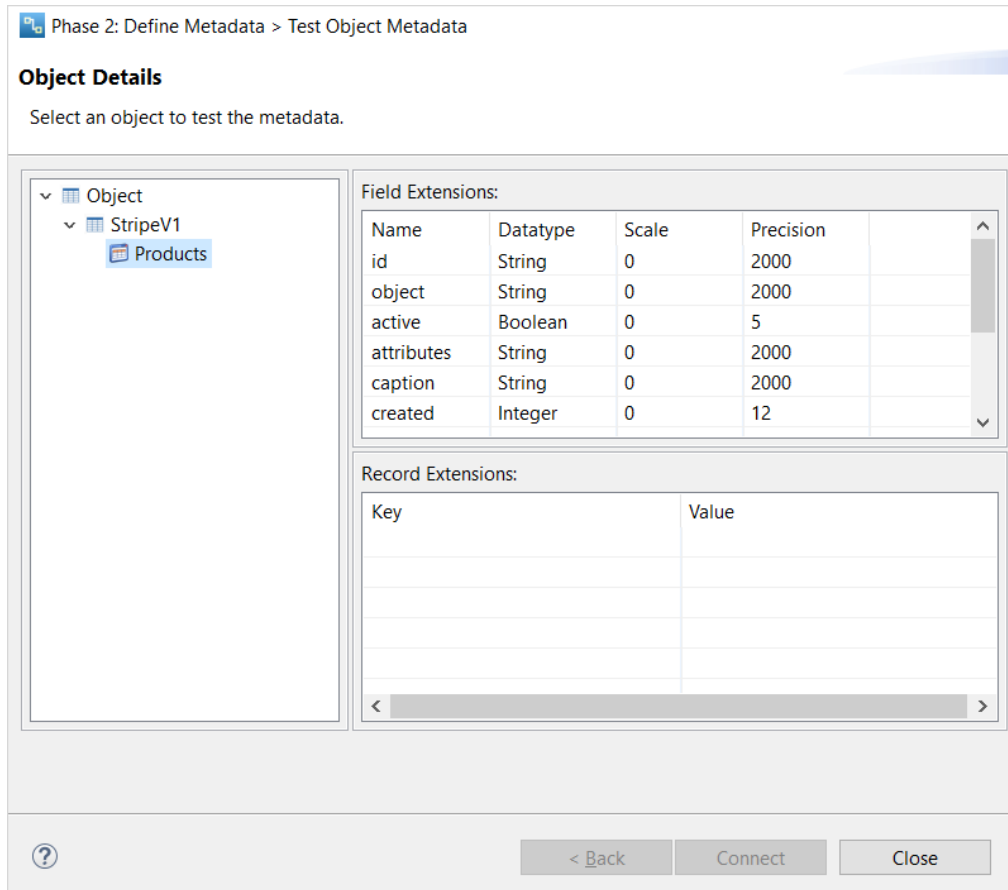
1. Click **Test Object Metadata** for the object.
2. Specify the JVM parameters for the virtual machine to debug the connector.



3. Click **Next**.
4. Enter the connection details.



5. Click **Connect**.
6. Select an object to test the metadata.



7. Click **Connect**.

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
```

Phase 3: Configure the runtime 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.

Set up runtime

Configure the runtime behavior of the connector in Java.

1. In the **Connector Progress** view, under **Phase 3: Configure the Runtime Behavior**, click **Set Up**.
The `<ConnectorID><NMOName>DataAdapter.java` and `<ConnectorID>DataConnection.java` files are generated.
2. 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`
3. Implement the following methods in the `<ConnectorID>DataConnection.java` file:
 - `connect`
 - `disconnect`

Test the read capability

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.

1. Under **Test Read Capability**, click **Test Runtime Behavior**.
The default JVM environment settings and tracing level are displayed. 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**.

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 handling 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.createNlsAdapterSDKException(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:

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

Implement runtime 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.

Phase 4: Generate the connector package

Use the Informatica Connector Toolkit to generate the connector package.

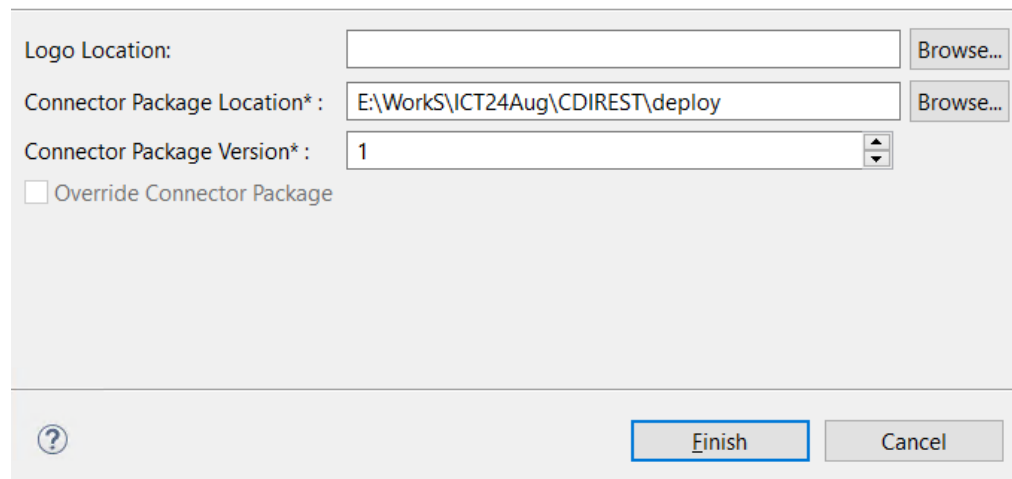
1. In the **Connector Progress** view, under **Phase 4: Generate the Connector Package**, click **Generate Package**.

The Generate Connector Package window appears.

 Phase 4: Generate Connector Package

Connector Package Settings

Specify the settings to generate the connector package.



The screenshot shows a dialog box titled "Phase 4: Generate Connector Package" with a sub-header "Connector Package Settings". Below the sub-header is the instruction "Specify the settings to generate the connector package." and a small icon of a document with a green checkmark. The dialog contains the following fields and controls:

- Logo Location:** A text input field with a "Browse..." button to its right.
- Connector Package Location* :** A text input field containing the path "E:\WorkS\ICT24Aug\CDIREST\deploy" with a "Browse..." button to its right.
- Connector Package Version* :** A spin box containing the value "1".
- Override Connector Package**
- At the bottom left is a question mark icon.
- At the bottom right are two buttons: "Finish" and "Cancel".

- Specify the following details to generate the connector package:

Property	Description
Logo Location	The image file for the logo of the connector. The file must be in PNG or JPEG format and the file size must not exceed 64 KB. The dimensions of the logo must be 89 x 45 pixels. The file name must not contain spaces or special characters.
Connector Package Location	The location where you want to save the connector package.
Connector Package Version	The version of the connector package.
Override Connector Package	Select to override the existing connector package.

- Click **Finish**.
If the connector package is generated successfully, a confirmation message is displayed.
- Click **OK**.

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

If you encounter an error or fail to publish the connector, refer to the `<connector_name>_codebuilder.log` file for details of the error. 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-DLT

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

Phase 5: Generate and run the test scenarios

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.

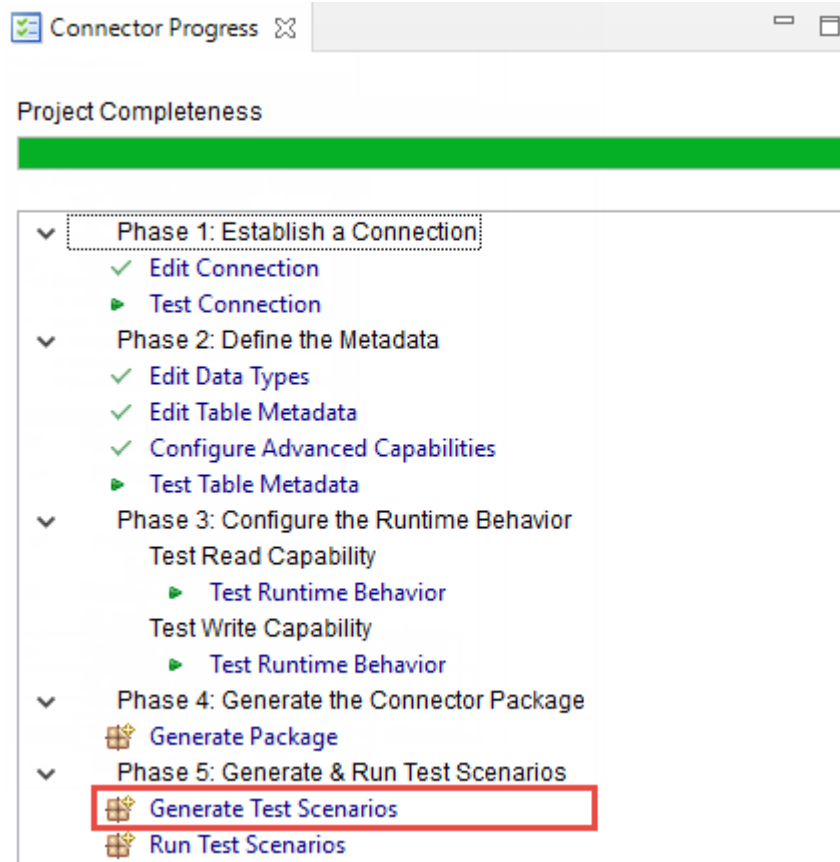
When the automation framework runs, it performs the following tasks:

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.

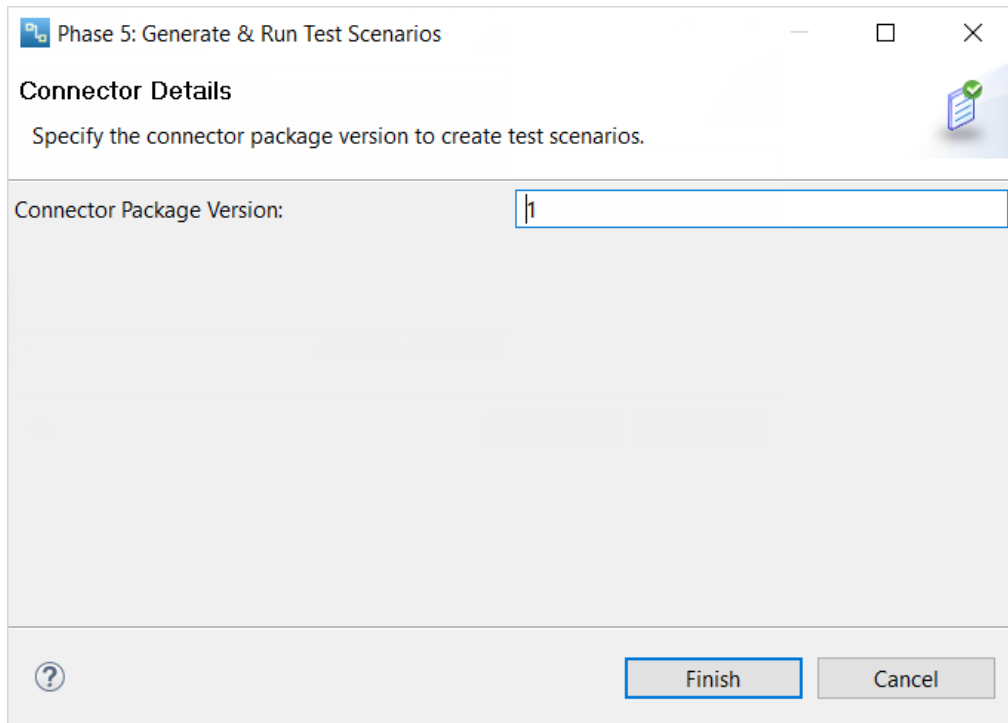
Generate the test scenarios

After you export the connector, you can create test scenarios for unit and integration testing.

1. In the **Connector Progress** view, under **Phase 5: Generate & Run Test Scenarios**, click **Generate Test Scenarios**.



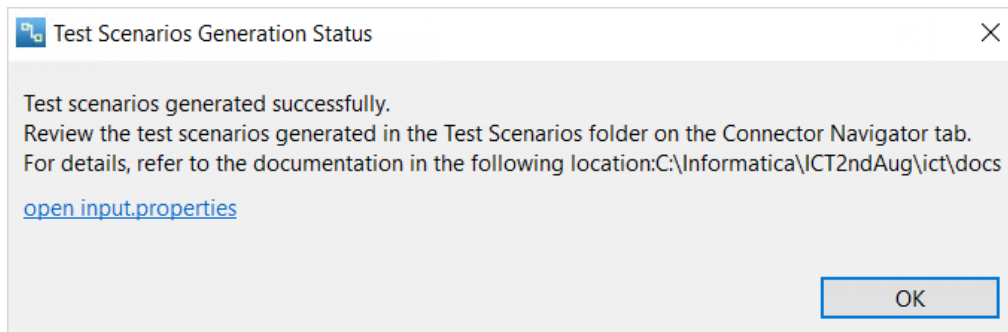
2. Specify the connector package version.



Ensure that you specify the same version number that you provided in Phase 4 before you generate the connector package.

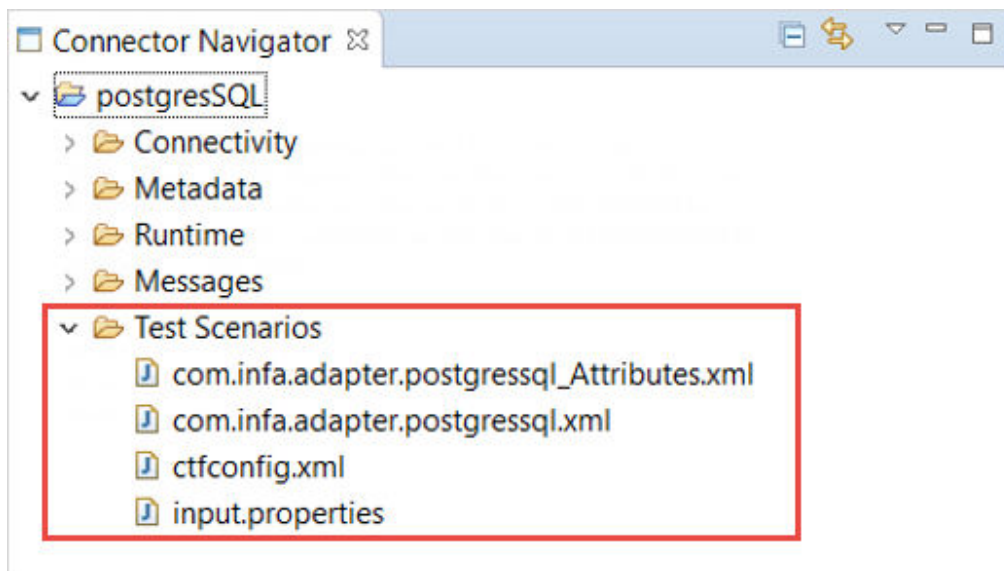
3. Click **Finish**

A confirmation message appears.



4. Click **OK**.
5. On the **Connector Navigator** tab, navigate to the **Test Scenarios** folder for the connector. The following files are generated in the Test Scenarios folder:
 - Attributes file
 - TestNG suite file
 - ctfconfig.xml file

- input.properties file



The following table describes the files generated in the test scenarios folder:

Files	Description
Attribute File	Generated as <code>com.infa.adapter.<adapter_name>_Attributes.xml</code> in the Test Scenarios folder. It consists of capability attributes such as read, write, and connection attributes. Specify the values for these properties.
TestNG Suite File	Generated as <code>com.infa.adapter.<adapter_name>.xml</code> in the Test Scenarios 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 <code>testsuite</code> file to add the actual values for the parameters. - Specify the values in the <code>input.properties</code> file.
<code>ctconfig.xml</code>	File that contains the required entries to run the test cases.
<code>input.properties</code>	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.

6. Open the TestNG Suite file.

A new tag named SANITY is added and the CAT and PAT tags are renamed to UNIT and INTEGRATION respectively.

```
--><test name="test_connection">
  <parameter name="tags" value="UNIT, INTEGRATION, SANITY"/>
  <parameter name="ConnectionName" value="connection1"/>
  <classes>
    <class name="com.informatica.cci.direct.CCITestConnection"/>
  </classes>
</test>
```

Configure parameters in 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.

```
61
62<test name="basic_read">
63  <parameter name="tags" value="CAT"/>
64  <parameter name="ConnectionName" value="connection1"/>
65  <parameter name="ReaderProperty" value="read1"/>
66  <parameter name="objectName1" value="$_EP{READ.objectName1}"/>
67  <classes>
68    <class name="com.informatica.cci.direct.CCIRuntimeReaderAllExpr"/>
69  </classes>
70 </test>
71
```

You can perform one of the following tasks:

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

```
61
62<test name="basic_read">
63  <parameter name="tags" value="CAT"/>
64  <parameter name="ConnectionName" value="connection1"/>
65  <parameter name="ReaderProperty" value="READ"/>
66  <parameter name="objectName1" value="public/employee"/>
67  <classes>
68    <class name="com.informatica.cci.direct.CCIRuntimeReaderAllExpr"/>
69  </classes>
70 </test>
```

- 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:

```
21#### Source Object for Reader testcases ####
22READ.objectName1=public/employee
23
```

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.

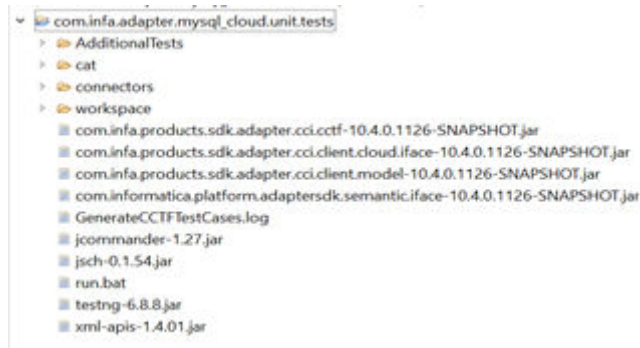
Node	Content
xml	version="1.0" encoding="UTF-8" standalone="no"
adapterAttribute	
connection	
name	connection1
user	
UIType	TEXTBOX
displayName	user
isHidden	false
type	STRING
value	\$_EP(CONNECTION.user)
attributeConfigDetails	

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** view, 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.

Run the test scenarios

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

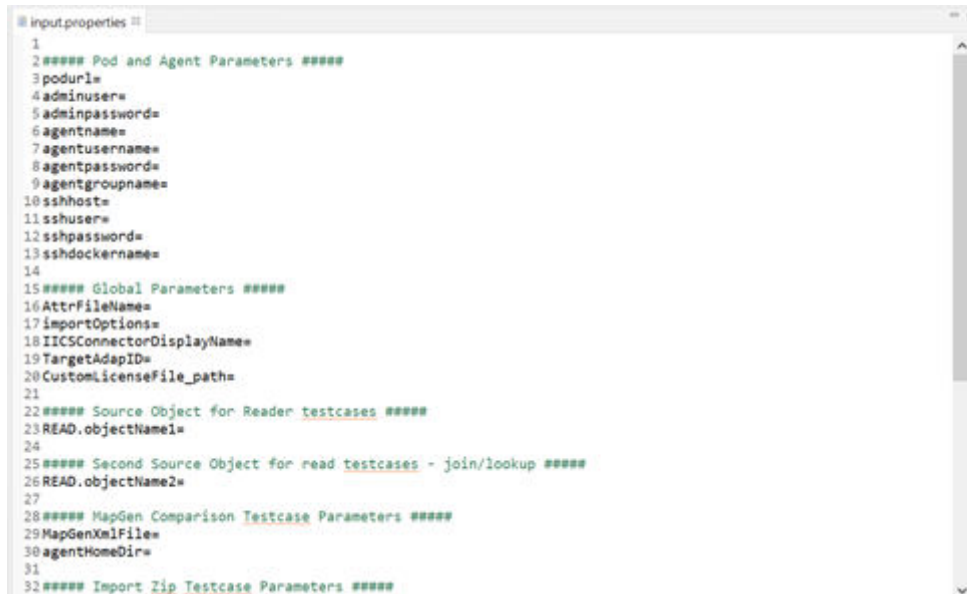
You can then view the success or failure status in the test report.

Unit tests and sanity tests

Perform the following steps to run the unit tests and sanity tests:


1. In the **Connector Navigator** view, navigate to the **Test Scenarios** folder for the connector.
2. Open the `input.properties` file and specify the parameters to execute the test scenarios.

The file contains a list of 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.



```
1
2#### Pod and Agent Parameters ####
3podurl=
4adminuser=
5adminpassword=
6agentname=
7agentusername=
8agentpassword=
9agentgroupname=
10sshhost=
11sshuser=
12sshpassword=
13sshdockername=
14
15#### Global Parameters ####
16AttrFileName=
17importOptions=
18IICSConnectorDisplayName=
19TargetAdapID=
20CustomLicenseFile_path=
21
22#### Source Object for Reader testcases ####
23READ.objectName1=
24
25#### Second Source Object for read testcases - join/lookup ####
26READ.objectName2=
27
28#### MapGen Comparison Testcase Parameters ####
29MapGenXmlFile=
30agentHomeDir=
31
32#### Import Zip Testcase Parameters ####
```

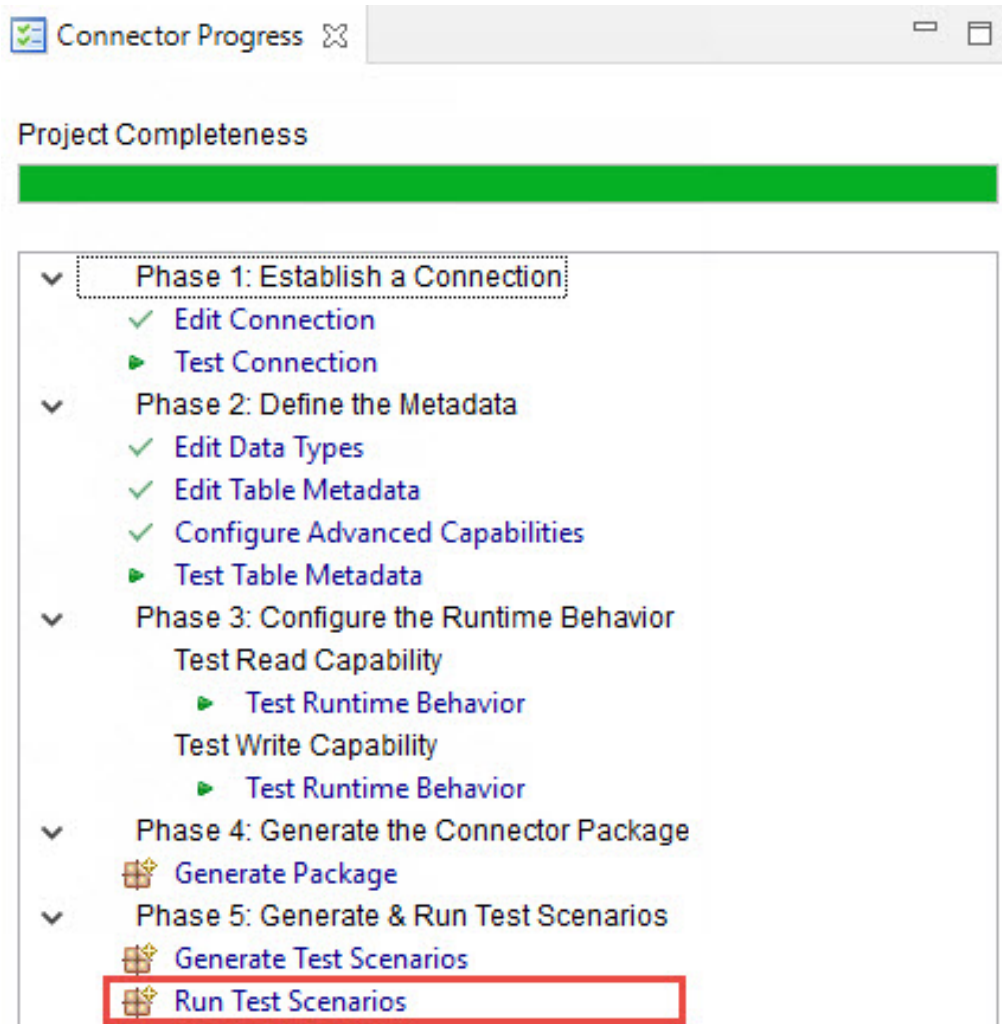
3. Edit the tag values in the `testNG` suite file. Specify `CAT` for Code Acceptance Test or Unit Test and `PAT` for Product Acceptance Test or Integration Test.



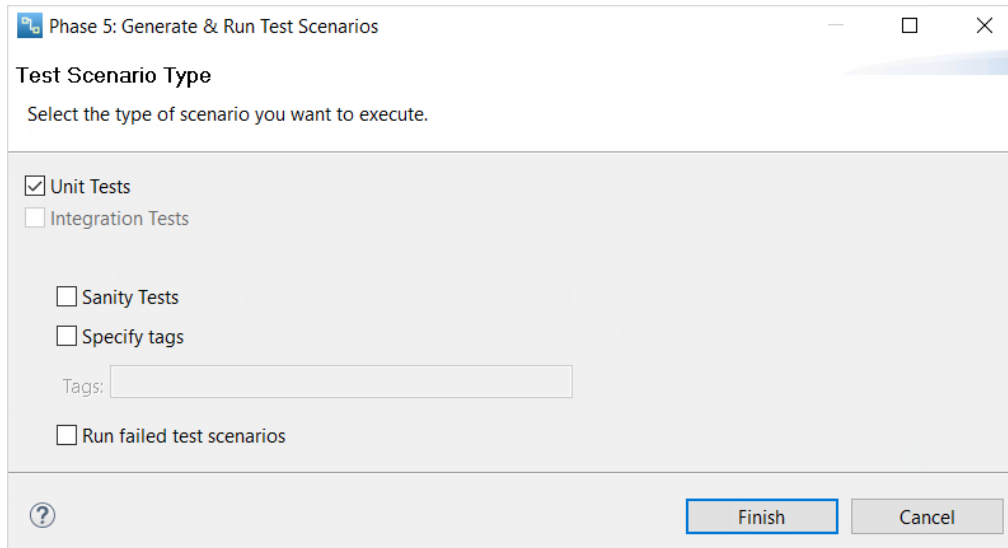
```
27<!--><test name="test_connection">
28  <parameter name="tags" value="CAT,PAT"/>
29  <parameter name="ConnectionName" value="connection1"/>
30  <classes>
31    <class name="com.informatica.cci.direct.CCITestConnection"/>
32  </classes>
33 </test>
34
35<test name="record_information">
36  <parameter name="tags" value="CAT"/>
37  <parameter name="ConnectionName" value="connection1"/>
38  <parameter name="objectName1" value="$_EP{READ.objectName1}"/>
39  <classes>
40    <class name="com.informatica.cci.direct.CCI_MetadataRecordInformation"/>
41  </classes>
42 </test>
43
44<test name="record_field_detail">
45  <parameter name="tags" value="PAT"/>
46  <parameter name="ConnectionName" value="connection1"/>
47  <parameter name="objectName1" value="$_EP{READ.objectName1}"/>
48  <classes>
49    <class name="com.informatica.cci.direct.CCI_MetadataRecordDetail"/>
50  </classes>
51 </test>
52
53<test name="record_search">
54  <parameter name="tags" value="NONE"/>
55  <parameter name="ConnectionName" value="connection1"/>
56  <parameter name="objectName1" value="$_EP{READ.objectName1}"/>
57  <classes>
58    <class name="com.informatica.cci.direct.CCI_MetadataSearch"/>
59  </classes>
60 </test>
61
```

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

4. In the **Connector Progress** view, under **Phase 5: Generate & Run Test Scenarios**, click **Run Test Scenarios**.

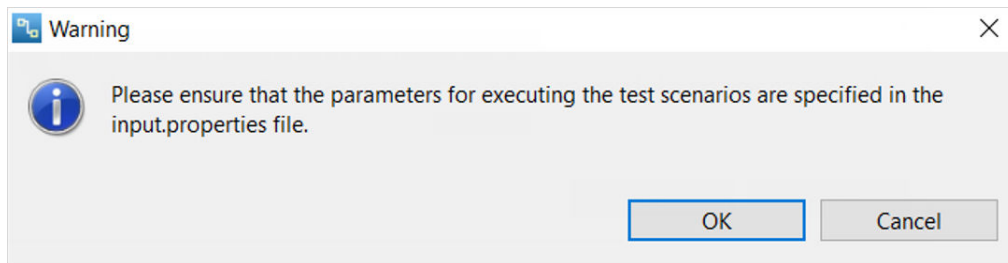


5. Select **Units Tests** and **Sanity Tests**.

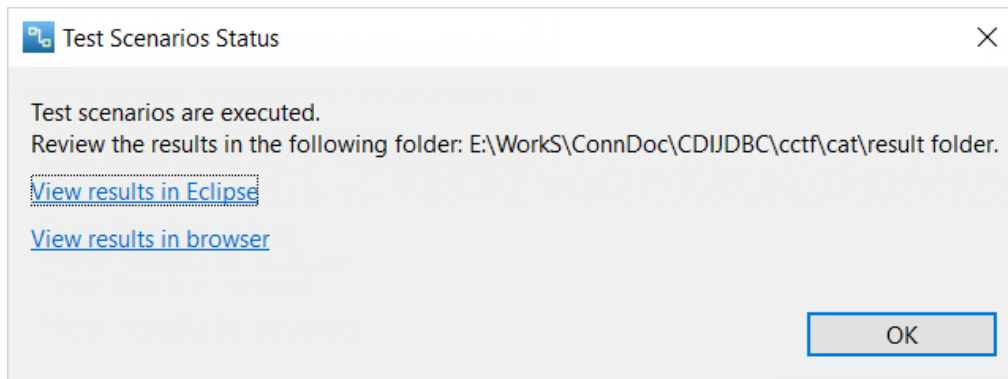


- Specify the tags to run specific unit tests.
- When you configure the **Units Tests** or **Sanity Tests** for the first time, the **Run failed test scenarios** option is not enabled.
After the test scenarios are executed, you can select the **Run failed test scenarios** option to execute only the failed test scenarios.
- Click **Finish**.

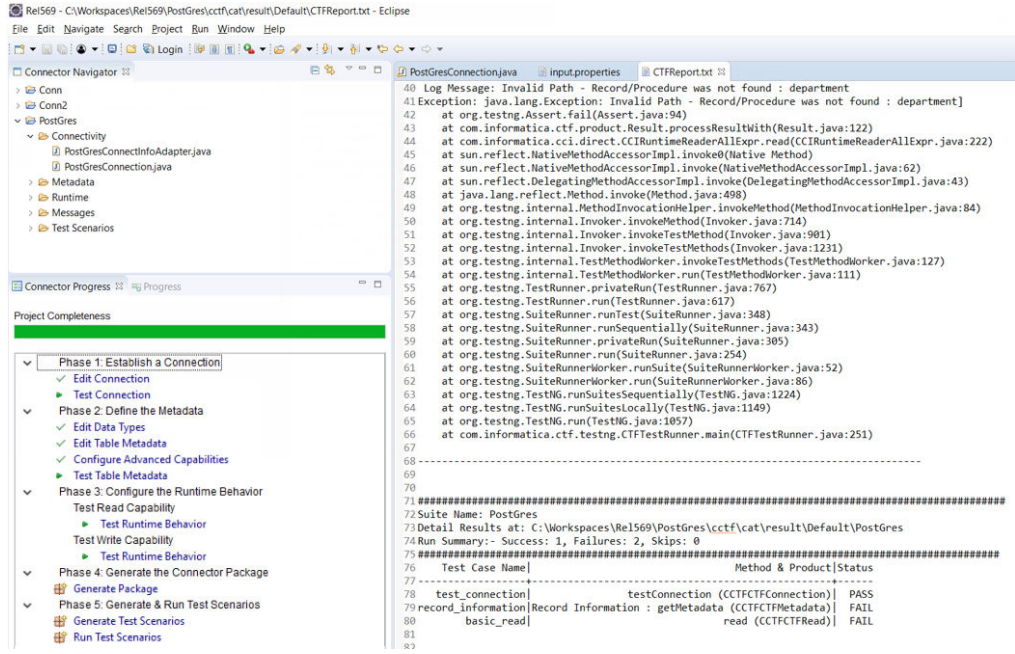
A warning window appears.



- Click **OK**.
After the test scenarios are executed, the **Test Scenarios Execution Status** window appears.



- Click **View results in Eclipse** to view the results in Eclipse.



Optionally, click **View results in browser** to view the results in the browser.

Test	# Passed	# Skipped	# Failed	Time (ms)	Included Groups	Excluded Groups
PostGres						
test_connection	1	0	0	5,831	SANITY	
record_information	0	0	1	200	SANITY	
record_field_detail	0	0	0	0	SANITY	
record_search	0	0	0	0	SANITY	
record_search_recordFilterType	0	0	0	0	SANITY	
record_search_recordFilterName	0	0	0	0	SANITY	
record_search_recordFilterDescription	0	0	0	0	SANITY	
record_search_recordFilterLabel	0	0	0	0	SANITY	
record_search_allFilter	0	0	0	0	SANITY	
basic_read	0	0	1	156	SANITY	
read_customBaselineTgt_filePath	0	0	0	0	SANITY	
read_portsTo_connect	0	0	0	0	SANITY	
mapGenXml_comparison	0	0	0	0	SANITY	
sessionLog_comparison	0	0	0	0	SANITY	
verifyMessage_for_testComparison	0	0	0	0	SANITY	
negative_testcase	0	0	0	0	SANITY	
dataVerification_skipNull	0	0	0	0	SANITY	
importZip_executeMCT	0	0	0	0	SANITY	
Total	1	0	2	6,187		

Class	Method	Start	Time (ms)
PostGres			
test_connection — passed			

11. Click **OK**.
12. To view the detailed log, see the `ctfrun.log` file in the result folder.
The folder also contains the `testing` and `junit` reports.

Integration tests

Perform the following steps to run the integration tests:

1. In the **Connector Navigator** view, navigate to the **Test Scenarios** folder for the connector and open the `ctfconfig.xml`.

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.

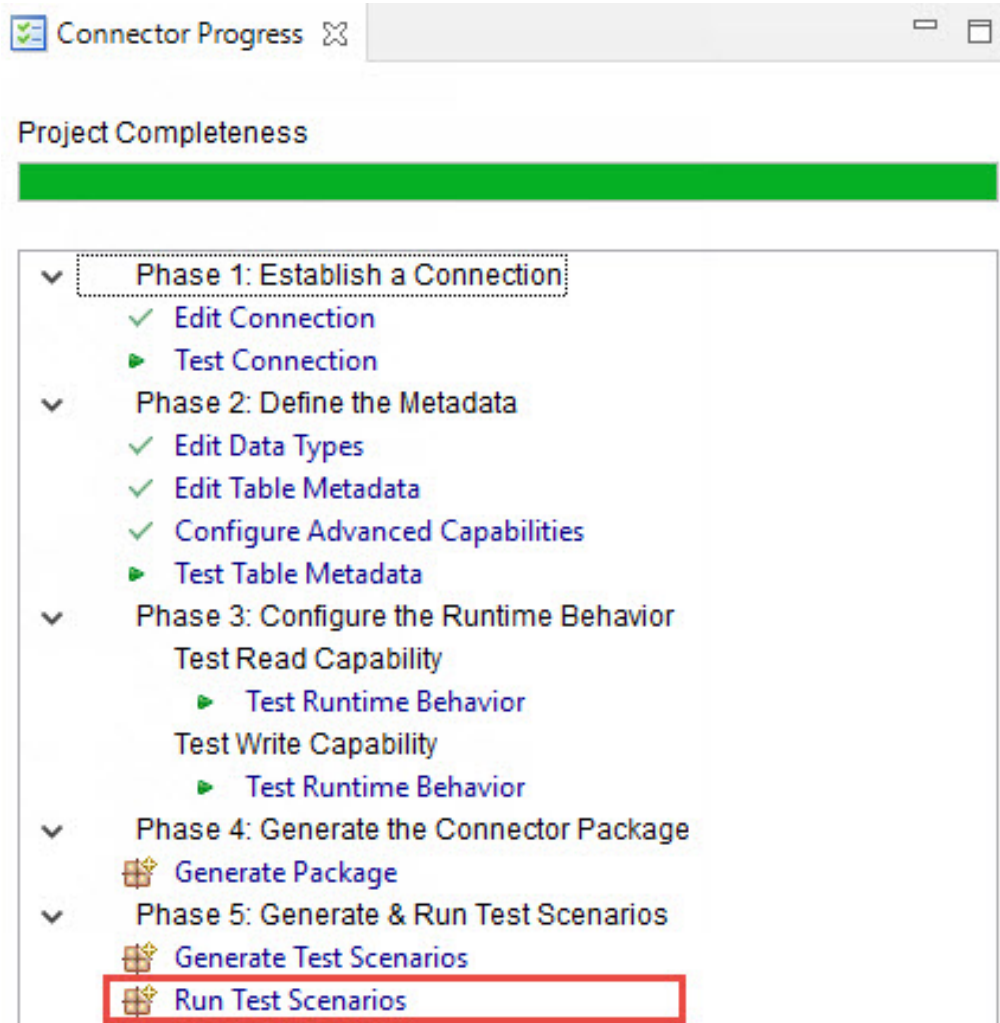
2. Specify the pod details in the `ctfconfig.xml` file or in the `input.properties` file.

Node	Content
xml	version="1.0" encoding="UTF-8" standalone="yes"
CTF	
Products	
CloudSecureAgent	
podURL	\$_EP(podurl)
adminUser	\$_EP(adminuser)
adminPassword	\$_EP(adminpassword)
agent	
name	\$_EP(agentname)
user	\$_EP(agentusername)
password	\$_EP(agentpassword)
groupName	\$_EP(agentgroupname)
sshHost	\$_EP(sshhost)
sshUser	\$_EP(sshuser)
sshPassword	\$_EP(sshpassword)
dockerName	\$_EP(sshdockername)
CCTF	
TestNGSuites	
resultLocation	result
Suite	
name	
file	com.infa.adapter.mysql_cloud.xml

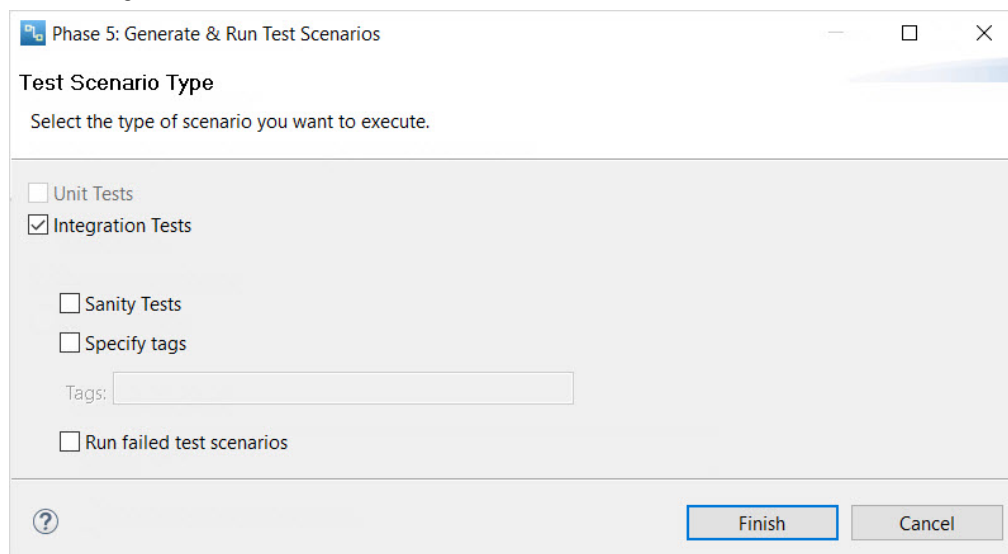
The following table describes the keys in the `ctfconfig.xml` file:

Key	Description
podurl	Pod URL for log in to the pod.
adminuser	Administrator user for the pod. Specify this parameter 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.
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 file.
Specify CAT for Code Acceptance Test or Unit Test and PAT for Product Acceptance Test or Integration Test.
Set the value of the tags to `NONE` during the test case review to skip the test case execution.
4. In the **Connector Progress** view, under **Phase 5: Generate & Run Test Scenarios**, click **Run Test Scenarios..**

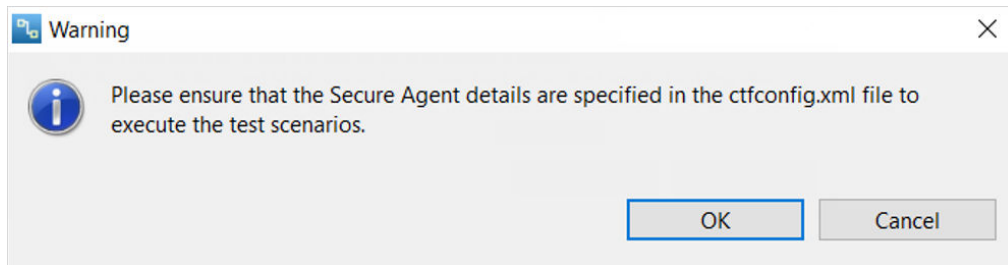


5. Select **Integration Tests**.

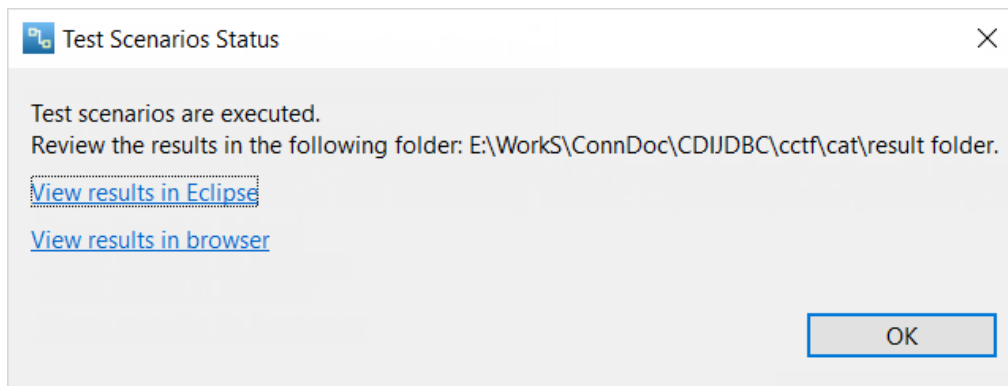


6. Specify the tags to run specific integration tests.

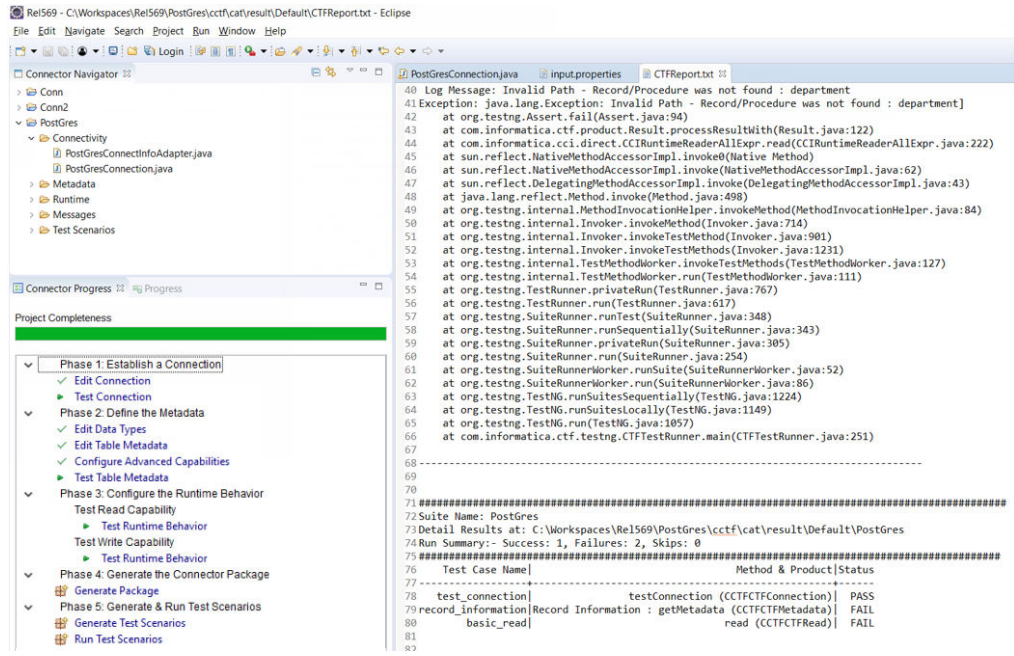
7. When you configure the **Integration Tests** for the first time, the **Run failed test scenarios** option is not enabled.
After the test scenarios are executed, you can select the **Run failed test scenarios** option to execute only the failed test scenarios.
8. Click **Finish**.
A warning window appears.



9. Click **OK**.
After the test scenarios are executed, the **Test Scenarios Execution Status** window appears.



10. Click **View results in Eclipse** to view the results in Eclipse.



Optionally, click **View results in browser** to view the results in the browser.

The screenshot shows a web browser displaying a TestNG Report. The browser address bar shows the file path: 'C:/Workspaces/Rel569/PostGres/cctf/cat/result/Default/emailable-report.html'. The report table shows test results for PostGres, with a total of 1 passed and 2 failed tests.

Test	# Passed	# Skipped	# Failed	Time (ms)	Included Groups	Excluded Groups
PostGres						
test_connection	1	0	0	5,831	SANITY	
record_information	0	0	1	200	SANITY	
record_field_detail	0	0	0	0	SANITY	
record_search	0	0	0	0	SANITY	
record_search_recordFilterType	0	0	0	0	SANITY	
record_search_recordFilterName	0	0	0	0	SANITY	
record_search_recordFilterDescription	0	0	0	0	SANITY	
record_search_recordFilterLabel	0	0	0	0	SANITY	
record_search_allFilter	0	0	0	0	SANITY	
basic_read	0	0	1	156	SANITY	
read_customBaselineTgt_filePath	0	0	0	0	SANITY	
read_portsTo_connect	0	0	0	0	SANITY	
mapGenXml_comparison	0	0	0	0	SANITY	
sessionLog_comparison	0	0	0	0	SANITY	
verifyMessage_for_testComparison	0	0	0	0	SANITY	
negative_testcase	0	0	0	0	SANITY	
dataVerification_skipNull	0	0	0	0	SANITY	
importZip_executeMCT	0	0	0	0	SANITY	
Total	1	0	2	6,187		

Class	Method	Start	Time (ms)
PostGres			
test_connection — passed			

11. Click **OK**.
12. To view the detailed log, see the `ctfrun.log` file in the result folder.
The folder also contains the `testng` and `junit` reports.

CHAPTER 5

Import a connector

To migrate a connector from one workspace to another, you can import the connector to the Eclipse IDE workspace.

Perform the following steps to import a connector:

1. Open the Eclipse workspace where you want to import the connector package.
2. Click **Window > Perspective > Open Perspective > Other**.
3. In the **Open Perspective** window, select **Informatica Connector**.
4. On the Eclipse IDE toolbar, click **Create New Connector**.
5. In the **Create New Connector Project** window, configure the following properties:

Property	Description
Connector Name	Name of the connector. Specify a unique name for the connector. The connector name must be an alphanumeric string. The first character of the name must be a letter.
Description	Description of the connector.
Cloud Service	The cloud service for which you want to create a connector. Select Cloud Data Integration .
Creation Method	The method to create a connector. Select one of the following creation methods: <ul style="list-style-type: none">- JDBC. Create a JDBC driver-based connector.- Libraries. Create a connector using a third-party library SDK.- REST. Create a connector for REST-based application. Default is REST.
Authentication Type	The authentication type for the REST creation method. Select one of the following authentication types: <ul style="list-style-type: none">- No Authentication: Does not require authentication.- Basic. Uses the 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. Default is basic authentication.

Note: Ensure that the connector name, cloud service, creation method, and authentication for the new connector project exactly match those of the existing connector that you want to import.

You can find the connector details for the existing connector in the adapterdefinition.xml file in following path:

<Connector package directory>/ict_metadata/adapterdefinition.xml

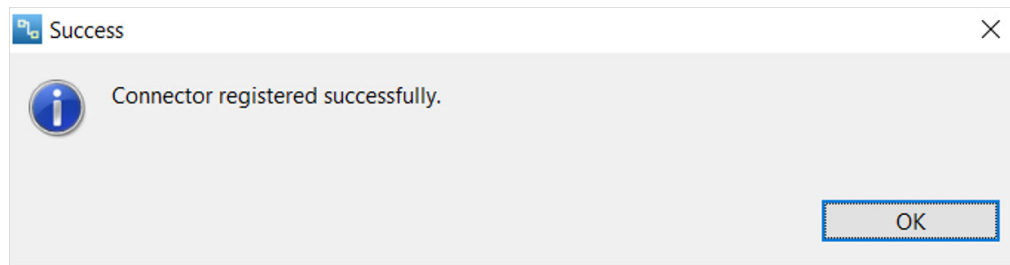
The following image shows the connector details in the adapterdefinition.xml file:

```

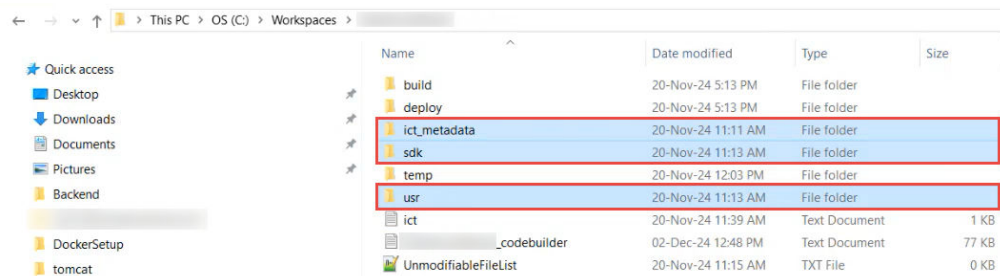
1 <Connector definitionVersion="10.0">
2   <ConnectorDefinition productType="DLT" patternSubType="Applications" authenticationType=
3     "OAuth_2.0" modelVersionNumber="6" uid="
4     connectorCategory="Database" connectorShortName="
5     "ENTERPRISE_APP" version="1.0.0" poolType="NONE" vendorShortName="INFA" vendorLongName=
6     "Informatica" name="
7     <Connections>
8       <Connection>
9         <name>
10        </name>
11        <endpointName></endpointName>
12        <ConnectionAttributes>
13          <ConnectionAttribute cliDisplayName="Authentication" defaultDisplayName=
14            "Authentication" description="" defaultValue="AuthorizationCode" isMandatory=
15            "true" isEncrypted="false" maxOccurs="0" minOccurs="0" persistSize="0"
16            maxLength="255" minLength="0" type="STRING" name="Authentication"
17            hasDependentAttr="true" isHidden="false" isSubstitutable="false"
18            isParameterizable="false">
19            <valuesList value="AuthorizationCode" displayValue="AuthorizationCode"/>
20            <valuesList value="PasswordCredentials" displayValue="PasswordCredentials"
21            />
22          </ConnectionAttribute>
23          <ConnectionAttribute cliDisplayName="AuthorizationURL" defaultDisplayName=
24            "AuthorizationURL" description="" defaultValue=

```

6. Click **Finish**.
If the connector is successfully registered, a success message appears.

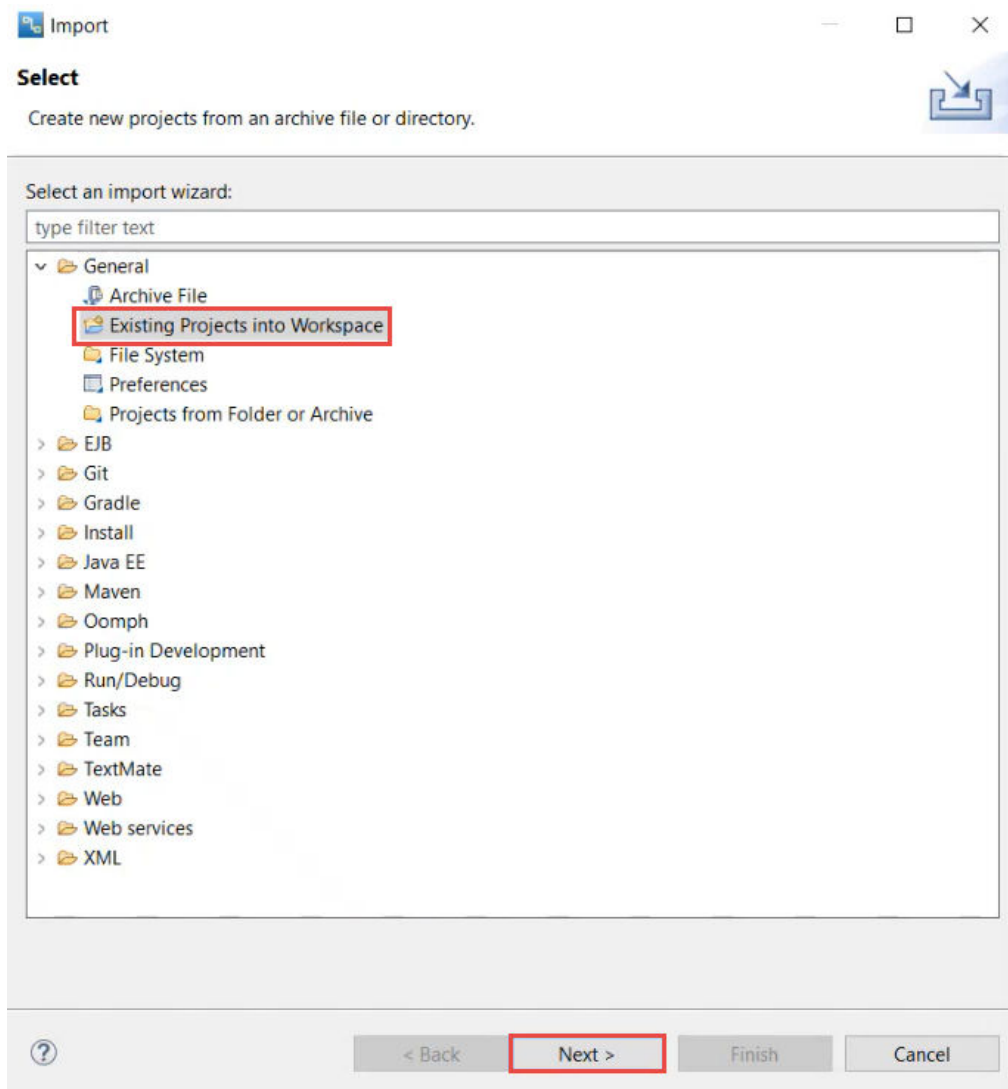


7. Click **OK**.
8. Open Windows explorer and navigate to the existing connector folder in the following path:
<old workspace>/<existing connector package folder>
9. Copy the ict_metadata, sdk, and usr folders.

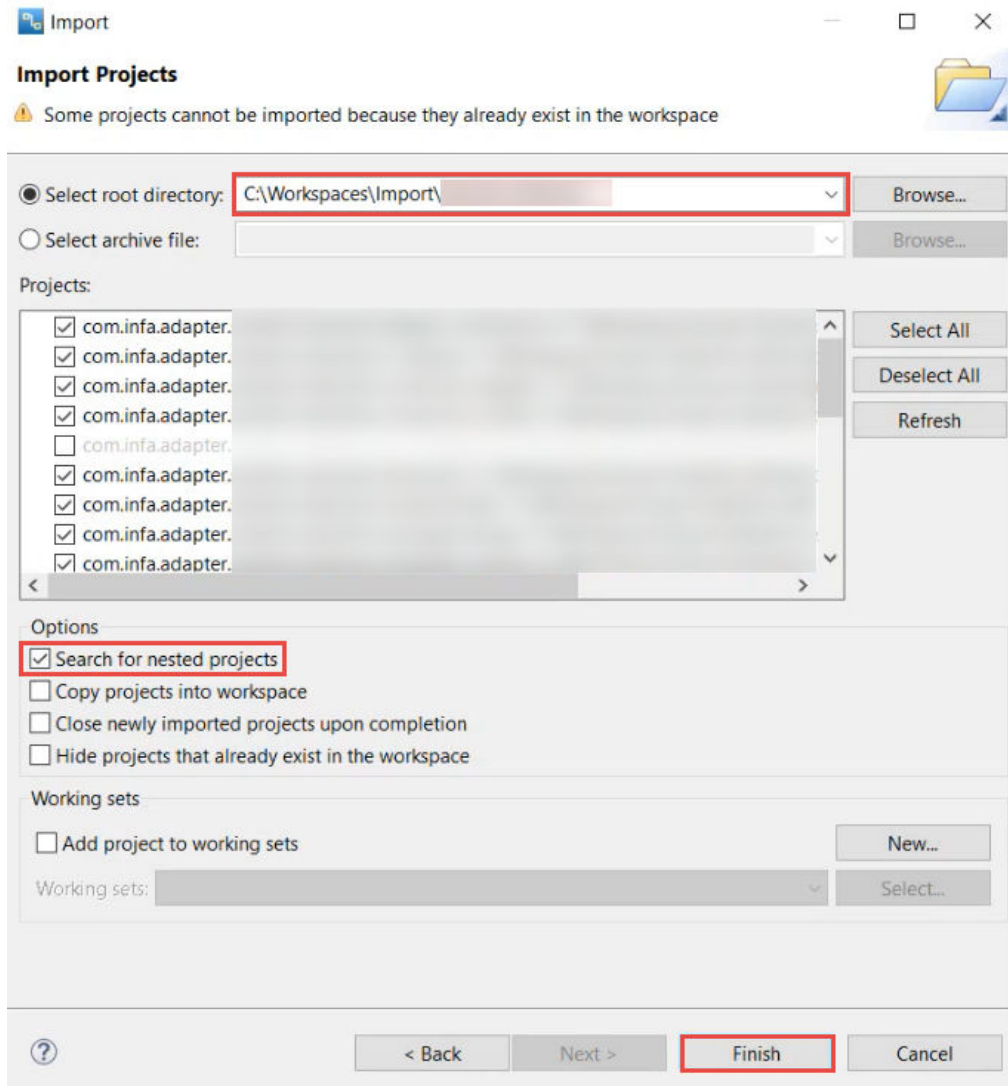


10. Navigate to the new connector folder in the following path:
<new workspace>/<new connector package folder>
11. Paste the copied folders.
When prompted with the option to replace or skip files, select **Replace the files in the destination**.
12. Close and re-open Eclipse IDE.
13. Click **File > Import**.

14. In the **Import** window, click **General > Existing Projects into Workspace**.



15. Click **Next**.
16. In the **Select root directory** field, browse and select the new connector folder.



17. Under **Options**, select **Search for nested projects**.

18. Click **Finish**.

19. In the **Connector Progress** view, under **Phase 1: Establish Connection**, click **Edit Connection**.

Note: If you have added any third-party libraries in the existing connector, remove them and add them again.

20. Click **Generate Code**.

Note: When you generate the code, certain Java classes are regenerated and any manual changes done in the code are overwritten.

CHAPTER 6

Connection attributes

The 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.

Attribute Property	Description
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 7

Type system

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 8

Connector metadata

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.
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 9

Partitioning capability

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 an 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 an 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 10

Pushdown capability

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 11

Mappings in advanced mode

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.mysql.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.ASOCComplexType
import
com.informatica.sdk.adapter.metadata.capabilityattribute.semantic.iface.ReadCapabilityAt
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.Session
import org.apache.spark.sql.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 MSQLEDataFrameAdapter 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 12

Manual changes to Informatica Connector Toolkit source code

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).isEmpty
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

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 = sdkFactory.newField(catalog);
    field.setNativeName(columnName);
    field.setScale(scale);
    field.setDataTypes(colType);
    field.setPrecision(precision);
    record.addField(field);
}
}
catch (SQLException e) {
    ExceptionManager.createNonNlsAdapterSDKException(
        "An error occurred 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 occurred while executing custom query for validation:["
+ e.getMessage() + "]);
        return false;
    }
    return true;
}

```

Code changes for object path override

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_CLOUDLIBRARYINFO");
    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 overridden
```

```
parameter value as the value.  
//Check if the overridden 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_ASOperation)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 finalOptimizedQuery=(String)pdoSQL.getOptimizedSql().get(0);
pushdownMetadata.setSql(finalOptimizedQuery);
if(finalOptimizedQuery.isEmpty()||finalOptimizedQuery==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(EIUDIIndicator.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_ASOperation
**/
@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.StructuralFeature;
import
com.informatica.sdk.adapter.metadata.projection.sinkoperation.semantic.iface.PlatformSink;
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 table names and ?columnname?
for column names. Users can provide a input port from the source as a parameter.
Connector needs to replace the values for these parameters before query execution.
 * SQLException is always the first field in SQL Tx. SQLException field is null/empty if
query is executed successfully.
 * If user selects RowsEffectted checkbox in SQL Tx, then RowsEffectted will be the
second field in SQL Tx. Connector developer should populate this field for every row
based on the number of rows effectted 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 SQLException 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 (SQLException 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,
+ dataType + "] "
                                "Data for column [" + columnName + "] of type ["
                                + "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,
+ dataType + "] "
                                "Data for column [" + columnName + "] of type ["
                                + "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,
+ dataType + "] "
                                "Data for column [" + columnName + "] of type ["
                                + "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,
+ dataType + "] "
                                "Data for column [" + columnName + "] of type ["
                                + "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;
    }
}

```



```

(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
SDKEException{
        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") ||
dataTableType.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,
+ dataType + "] "
                    "Data for column [" + columnName + "] of type ["
                    + "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,
+ dataType + "] "
                    "Data for column [" + columnName + "] of type ["
                    + "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,
+ dataType + "] "
                    "Data for column [" + columnName + "] of type ["
                    + "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);
    }
}

```


CHAPTER 13

Runtime behavior

Use the functions in the Informatica Connector Toolkit API to specify the runtime 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.

Runtime 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 14

Connector example: MySQL_Cloud

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 15

Version control integration

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

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:

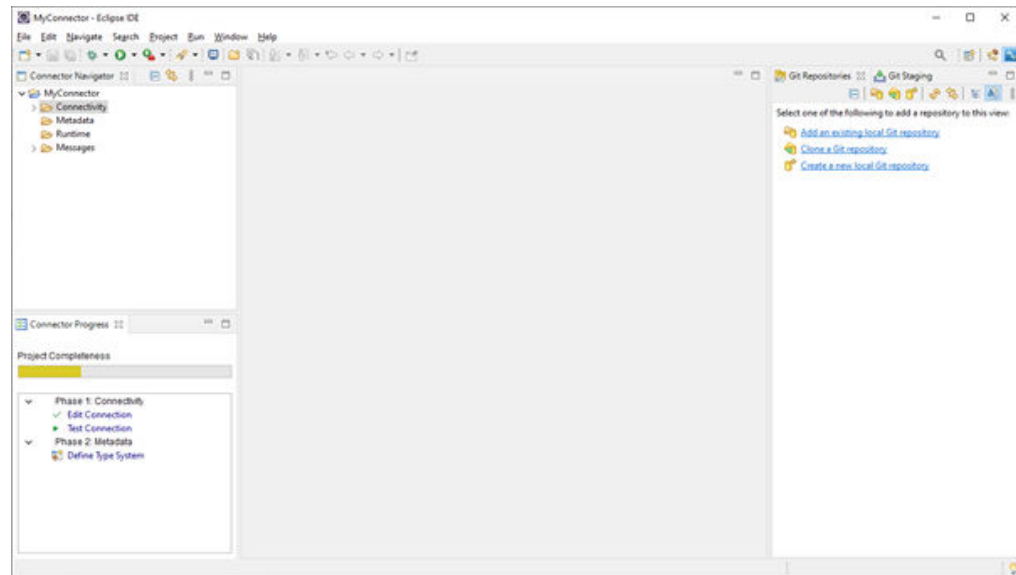
- Install Git version 2.33.
- Set the custom property `UseGit=true` in the `env.properties` file in the following directory:
`<eclipse_home>/dropins/com.informatica.tools.ui.ict`
- If Eclipse is already running, restart Eclipse.

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:

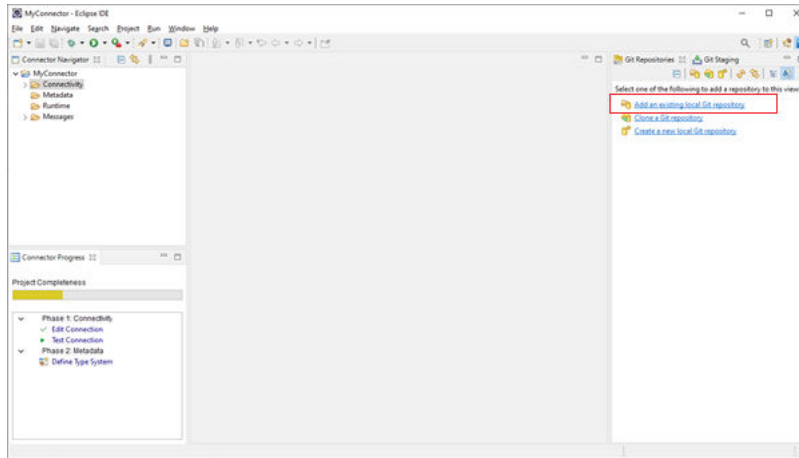


7. Follow the instructions from Chapter 3 to create a new connector.

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

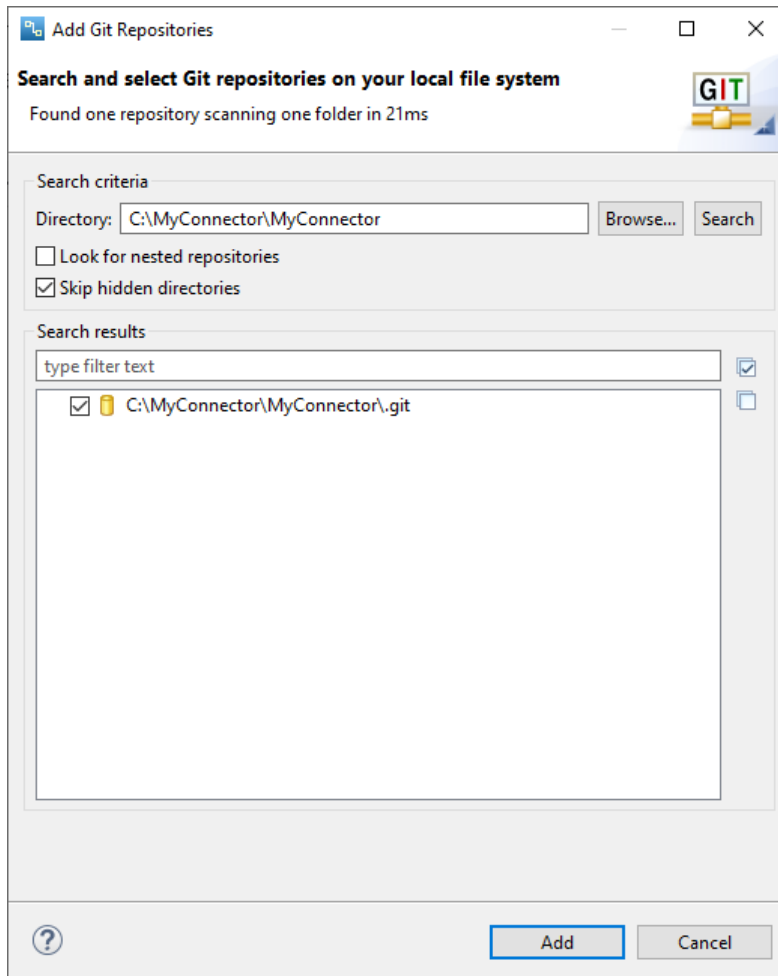
Name	Date modified	Type	Size
.git	25-02-2022 03:40 PM	File folder	
build	25-02-2022 03:40 PM	File folder	
ict_metadata	25-02-2022 03:39 PM	File folder	
sdk	25-02-2022 03:40 PM	File folder	
temp	25-02-2022 03:41 PM	File folder	
usr	25-02-2022 03:40 PM	File folder	
.gitignore	25-02-2022 03:40 PM	Text Document	1 KB
ict	25-02-2022 03:40 PM	LOG File	1 KB
MyConnector_codebuilder	25-02-2022 03:40 PM	LOG File	5 KB
UnmodifiableFileList	25-02-2022 03:40 PM	TXT File	0 KB

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



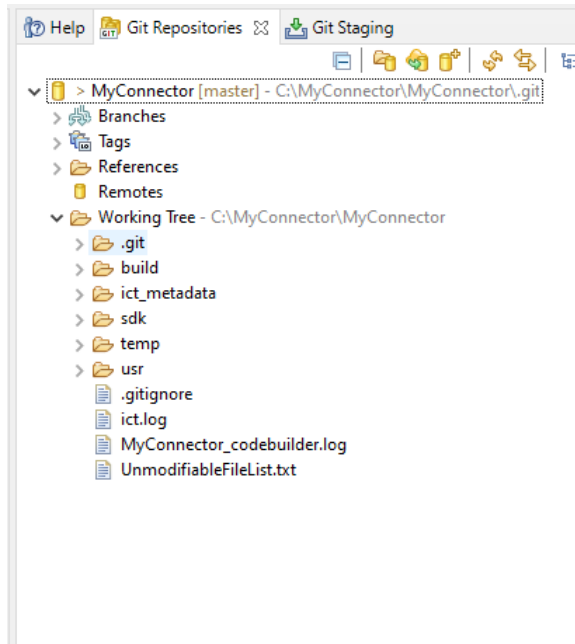
A search window appears.

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



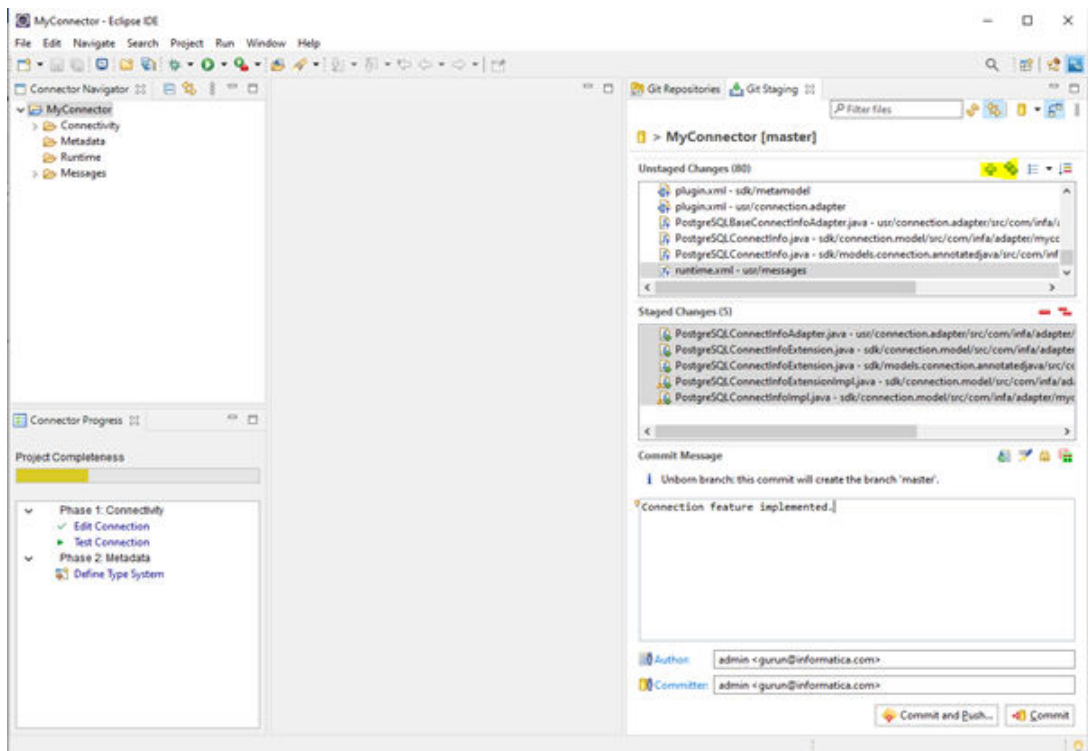
10. Select the file and click **Add**.

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 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.

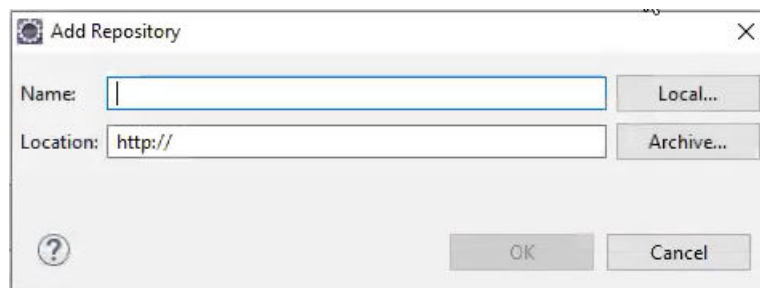
For example, enter the following location to install the Perforce Eclipse plugin for Eclipse 4.7:

<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**.

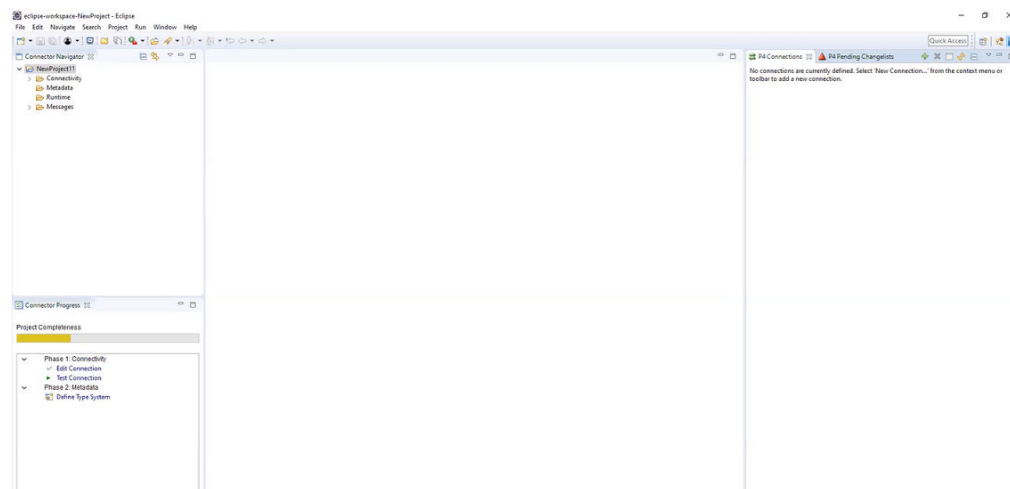
The **New Informatica Connector Project** page appears.

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:

Name	Date modified	Type	Size
build	05-04-2022 04:17 PM	File folder	
ict_metadata	05-04-2022 04:15 PM	File folder	
sdk	05-04-2022 04:17 PM	File folder	
usr	05-04-2022 04:16 PM	File folder	
ict	05-04-2022 04:17 PM	LOG File	1 KB
MyConnector1_codebuilder	05-04-2022 04:17 PM	LOG File	5 KB
p4ignore	05-04-2022 04:17 PM	TXT File	1 KB
UnmodifiableFileList	05-04-2022 04:16 PM	TXT File	0 KB

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.

- Specify the server details and the perforce user account details.
The following image shows the perforce server connection page:

Perforce Server Connection Wizard

Add a new Perforce server connection

Add a new Perforce server connection to the P4 Connections view

Location

Recent servers: [dropdown]

Server: Perforce:1666

Charset: none [dropdown]

Authentication

User: PerforceUser1

Password: [masked]

Store in Secure Store

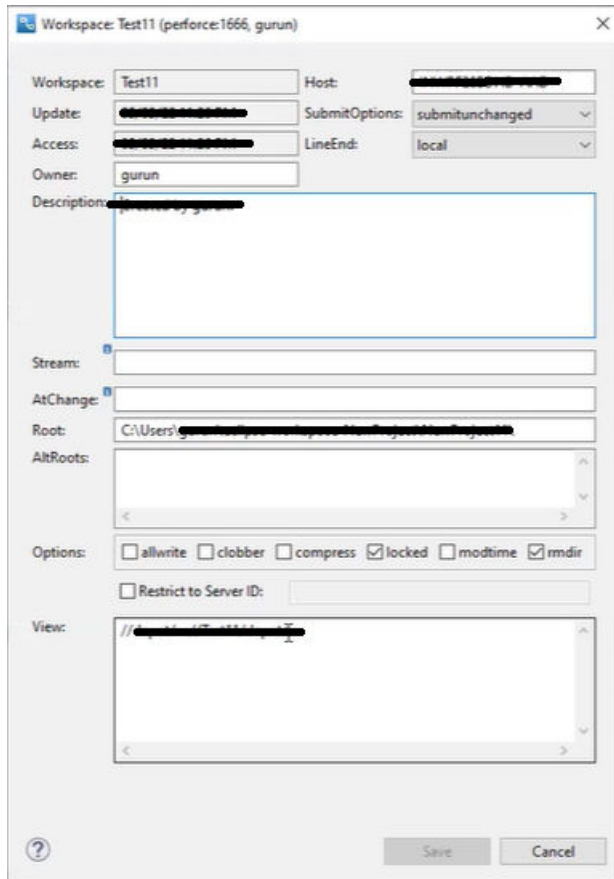
Ignore SSL Validation

[?] < Back Next > Finish Cancel

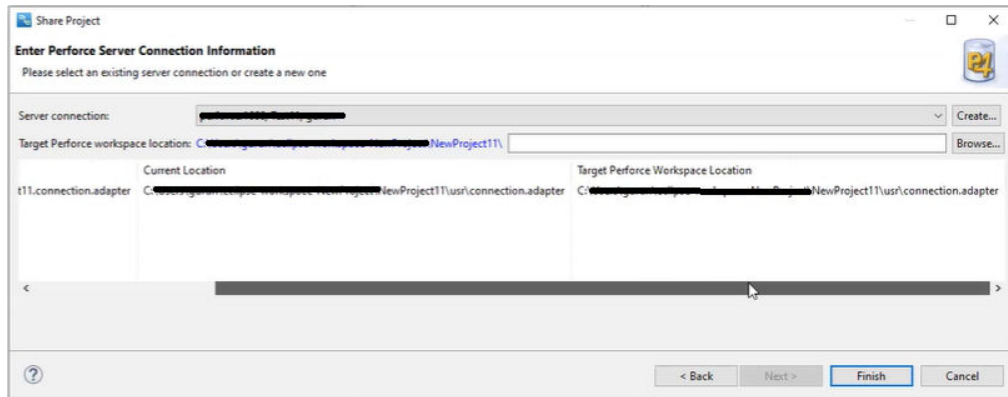
- Click **Next**.
- Select an existing Perforce workspace or create a new Perforce workspace.
To create a new Perforce workspace, enter the workspace name. Enter the location or browse to the eclipse workspace of the current project and select the folder.
- Clear the **Launch the Perforce Project Import Wizard** check box, and click **Finish**.
The connection is created.
- In the P4 Connections view, right click on the connection, and select **Edit Perforce Workspace**.
The **Workspace** window appears.
- 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:

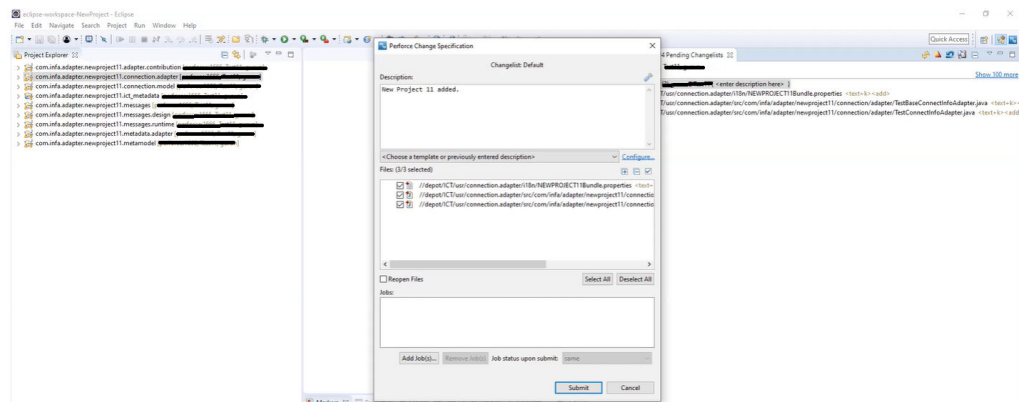


14. Click **Save**.
The **Perforce Workspace** window appears.
15. Select the **Do not synchronize the projects** check box, and click **OK**.
16. Select **Window > Perspective > Open Perspective > Java**.
17. In the **Project Explorer** pane, select all the Java projects, right click, and navigate to **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

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

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

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
<code>generateCode</code>	Generates the connector code and <code>build.xml</code> using the configurations defined in <code>ict_metadata.xml</code> .
<code>buildAdapter</code>	Compiles and deploys the connector project.
<code>backupProject</code>	Creates a backup file for the connector project.
<code>restoreProject</code>	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] [...]`

Migrate connector 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 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 to return 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 directTSMMap, 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

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

C

- connection
 - attributes [96](#)
 - create [96](#)
 - test [101](#)
- connection attributes
 - define [22](#), [96](#)
- connector
 - build [19](#), [93](#)
 - deploy [75](#), [114](#)
 - rest api [49](#)
 - third-party library [38](#)
- create
 - connector [19](#), [93](#)
 - messages [113](#)

D

- define
 - connection attributes [22](#), [96](#)
 - metadata [28](#)
 - record [30](#), [102](#)
 - type system [29](#), [102](#), [121](#)

E

- Eclipse
 - install [13](#)

I

- Informatica Connector Toolkit
 - install [14](#)
 - overview [9](#)
- Informatica Global Customer Support
 - contact information [8](#)
- install
 - Informatica Connector Toolkit [14](#)

M

- metadata
 - define [28](#), [30](#), [102](#)
- MySQL_Cloud connector
 - requirements [183](#)
- MySQL_Cloud Connector
 - components [184](#)

P

- publish
 - adapter [75](#), [114](#)

R

- rest api
 - flattened output [49](#)
 - hierarchical output [54](#)
- run time
 - overview [181](#)
- runtime
 - configure [69](#)
 - test [70](#)

T

- test
 - connection [27](#), [101](#)
 - metadata [66](#), [108](#)
 - read capability [70](#), [111](#)
 - write capability [71](#)
- test scenarios
 - generate [78](#), [117](#)
 - run [82](#)
- type system
 - define [29](#), [102](#), [121](#)