This guide describes how to create custom adapters using the Cloud Adapter SDK.
# Contents

Preface ............................................................................................................................................................... vii  
   Audience ...................................................................................................................................................... vii  
   Prerequisites ................................................................................................................................................ vii  
   Related Documentation and Resources ................................................................................................. vii  
   Conventions................................................................................................................................................ viii  

1 About Creating Custom Cloud Adapters with Cloud Adapter SDK

Custom Adapters ...................................................................................................................................... 1-1  
Cloud Adapter SDK ................................................................................................................................ 1-2  
Adapter Framework Components ........................................................................................................ 1-2  
Design-Time Functional Components ................................................................................................... 1-5  
Runtime Functional Components ........................................................................................................... 1-6  
Adapter Wizard Framework ................................................................................................................... 1-7  
Adapter Development Life Cycle........................................................................................................... 1-9  

2 Developing the Sample Cloud Adapter

Differences between Outbound and Inbound Adapters ................................................................. 2-1  
Setting Up the Cloud Adapter Development Environment....................................................... 2-1  
Creating the Adapter Project........................................................................................................ 2-2  
Creating a cloud-adapter.xml File for a Cloud Adapter............................................................ 2-8  
Design-Time Components.................................................................................................................. 2-11  
User Interface Module....................................................................................................................... 2-13  
   Adapter User Interface Entry Point ............................................................................................ 2-14  
   Welcome Page ........................................................................................................................... 2-20  
   Connection Page ....................................................................................................................... 2-23  
   Operations Page for Outbound Adapters .............................................................................. 2-25  
   Request Configuration Page for Inbound Adapters ........................................................... 2-28  
   Response Configuration Page for Inbound Adapters ........................................................ 2-30  
   Summary Page ........................................................................................................................ 2-33  
Plugin Module........................................................................................................................................ 2-37  
   Connection Properties and Testing ....................................................................................... 2-37  
   Metadata Browser ..................................................................................................................... 2-40
3 Developing the Sample REST Cloud Adapter
About REST Cloud Adapters................................................................. 3-1
Creating a Metadata Catalog for a REST Cloud Adapter................... 3-2
Setting Up the REST Cloud Adapter Development Environment ...... 3-2
Creating a cloud-adapter.xml File for a REST Cloud Adapter .......... 3-3
Creating Outbound REST Cloud Adapter Design-Time Components ... 3-4
Creating Outbound REST Cloud Adapter Runtime Components ........ 3-11
Creating Inbound REST Cloud Adapter Components ...................... 3-13

4 Building and Packaging the Sample Cloud Adapter
Building the Sample Cloud Adapter .................................................. 4-1
Packaging and Generating the JAR File ............................................. 4-2
Registering the Adapter................................................................. 4-6
Testing an Outbound Adapter....................................................... 4-7
  Creating an Application for Testing an Outbound Adapter .......... 4-7
  Adding an External Reference and Using the Adapter Wizard ..... 4-9
  Creating a BPEL Client to Invoke the Adapter ......................... 4-13
  Deploying the Application that Tests the Outbound Adapter ...... 4-17
  Testing the Outbound Adapter Using the EM Console .......... 4-19
Testing an Inbound Adapter............................................................ 4-20
  Creating an Application for Testing an Inbound Adapter ......... 4-21
  Adding an Exposed Service and Using the Adapter Wizard ... 4-23
  Creating a BPEL Process to Consume the Adapter ............. 4-27
  Deploying the Application that Tests the Inbound Adapter ...... 4-32
  Testing the Inbound Adapter Using the EM Console ......... 4-32

5 Cloud Adapter SDK Advanced Features
Design-Time Framework Description .................................................. 5-1
Batch and Bulk Operations.............................................................. 5-2
Response Mapping............................................................................. 5-5
Queries and DSLs ............................................................................ 5-5
Edit Mode........................................................................................... 5-6
Metadata Downloading and Caching............................................... 5-7
Error Handling Support for REST-Based Adapters ....................... 5-11
Runtime Framework Execution Flow .............................................. 5-12
## Cloud Adapter SDK Reference Information

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter Troubleshooting</td>
<td>A-1</td>
</tr>
<tr>
<td>JDeveloper Wizard Design Recommendations</td>
<td>A-2</td>
</tr>
<tr>
<td>Unified UI Framework Classes</td>
<td>A-3</td>
</tr>
<tr>
<td>Unified UI Field Object Names and Descriptions</td>
<td>A-10</td>
</tr>
</tbody>
</table>
Preface

*Developing Custom Oracle Cloud Adapters* describes how to create custom adapters using the Cloud Adapter SDK.

**Topics:**

- Audience
- Prerequisites
- Related Documentation and Resources
- Conventions

**Audience**

If you are a cloud adapter architect or developer, this book will teach you about Cloud Adapter SDK concepts, features, and development processes.

This book provides guidelines for custom adapter development using a sample adapter reference implementation.

Before reading this document, you should understand Java, web services, and integration concepts, including adapters.

**Prerequisites**

Install the required software as described in the Cloud Adapter SDK Installation Guide. The downloads you need are:


**Related Documentation and Resources**

Oracle provides a comprehensive set of preexisting adapters to connect with various data sources, including common and popular SaaS applications, enterprise applications, databases, messaging infrastructure systems, and other technologies.

For more information, see these Oracle resources:

- Getting Started with Oracle Cloud
• Oracle Integration Cloud Service for more documentation, videos, and tutorials, including information on adapters.

• Oracle Public Cloud

http://cloud.oracle.com

• Cloud Adapter SDK Installation Guide

Conventions

The following text conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><strong>monospace</strong></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
This chapter describes the basic components of the Cloud Adapter SDK.

Topics

- Custom Adapters
- Cloud Adapter SDK
- Adapter Framework Components
- Design-Time Functional Components
- Runtime Functional Components
- Adapter Wizard Framework
- Adapter Development Life Cycle

Custom Adapters

If no adapter exists for the application with which your application needs to integrate, then you, the developer, can create a custom inbound or outbound adapter using Cloud Adapter SDK.

When creating a custom adapter, you can:

- Create an adapter for a cloud, on-premises, or legacy application
- Connect securely, with authentication, authorization, and session management
- Provide support for audits, compliance, and governance

A well-designed custom adapter shields its users, such as integration modelers and business logic developers, from coding and from most application configuration details. Users see an intuitive wizard that provides a simplified view of an application’s business operations and business objects. Wizard users can browse and select one or more services or objects of interest and run the supported operations. An adapter contains general-purpose, reusable code, leaving the business logic to the target application.

An adapter invoking an application is called `outbound invocation`. The wizard user specifies a business document and the operation to be performed on the document. For example, the business document might be a sales order or an opportunity, and the operation might be to create or update the document.

An application invoking an adapter is called `inbound invocation`. The wizard user chooses a business event to start the integration flow. For example, the business event
might be receiving account details from Oracle Sales Cloud, and the integration flow might synchronize the account details with other applications.

**Cloud Adapter SDK**

The Cloud Adapter SDK is a set of Java APIs, tools, and implementation classes that you use to develop adapters.

Adapters built with the Cloud Adapter SDK are also referred to as cloud adapter plugins. Benefits are:

- Lets you build, test, and release custom adapters quickly.
- Provides many needed adapter capabilities as default behaviors.
- Parses standard service interfaces, builds a standard data model, and includes the model in the wizard to simplify configuration. Standard and custom business objects can be displayed differently in the wizard.
- Lets you build once and deploy anywhere. Adapters built once using the Cloud Adapter SDK are pluggable across integration platforms such as Oracle SOA Suite, Oracle Service Bus, and Integration Cloud Service (ICS).

**Adapter Framework Components**

The Adapter Framework has two main components: the Design-Time Framework and the Runtime Framework. The Design-Time Framework creates runtime artifacts that the Runtime Framework uses to perform the integration.

**Design-Time (DT) Framework**

At design time, the wizard user configures the adapter, selecting business operations and business objects, using the Adapter Configuration Wizard, also called the user interface (UI). The wizard uses the underlying Cloud Adapter SDK to connect to the application, browse the application metadata, and generate runtime artifacts.

Adapters that implement the DT framework are used in designer tools such as JDeveloper and Integration Cloud Service (ICS).

**Runtime (RT) Framework**

At runtime, this framework uses the runtime artifacts to invoke and exchange messages with the source or target application. The adapter can perform optional message transformations or use a specific message exchange pattern.

The following diagram illustrates the relationship between the Design-Time and Runtime frameworks:
The following diagram shows the wizard (or UI) pages, the required plugin classes developed using the Cloud Adapter SDK, and the runtime artifacts generated from wizard inputs.

The following diagram shows how the artifacts generated at design time are connected to the runtime framework classes developed using Cloud Adapter SDK.
The key runtime artifacts that the design-time framework generates are:

- **Integration WSDL**: The integration WSDL is based on the end user’s selection of specific business operations and business objects in the wizard. The integration WSDL is an interface between the integration platform and the application. It does not contain any concrete bindings or endpoints. The actual bindings and endpoints are defined in the Endpoint WSDL or WADL. The integration WSDL is generated for both inbound and outbound adapters.

- **Cloud Application Configuration File**: The integration platform uses this XML file to invoke an application instance. This file contains all the information the RT framework needs to invoke an application operation. This file is generated for both inbound and outbound adapters.

- **Endpoint WSDL**: For SOAP-based adapters, this is the service interface between the adapter and the application. Clients can submit requests to the adapter based on this service interface. The Endpoint WSDL is generated for an inbound adapter only.

- **Endpoint WADL**: For REST-based adapters, this is the service interface between the adapter and the application. Clients can submit requests to the adapter based on this service interface. The Endpoint WADL is generated for an inbound adapter only.

For outbound integration, the RT framework receives requests from the integration platform and translates these requests into request/response interactions with the application.

For inbound integration, the RT framework receives requests from the application and passes them to the integration platform.

To leverage the RT framework, adapters must implement an interface that the integration platform invokes. Once invoked, the interface delegates the request to the
application. As part of this invocation, the RT framework receives the payload, the generated integration WSDL, the name of the application operation that needs to be invoked, and any properties set by the DT framework, such as connection information.

The RT framework also performs translation, authentication, and authorization, and session management.

The following diagram shows the design time and runtime environments:

![Design-Time Functional Components Diagram]

**Design-Time Functional Components**

This section describes the design-time functional component modules, including the wizard, adapter plugin, connection module, metadata browser, data models, and runtime metadata generation.

**Wizard Modules**

You use and extend the Adapter Wizard Framework (AWF) in the Cloud Adapter SDK to collect user input in the wizard. The AWF is the DT adapter plugin driver.

**Adapter Plugin Module**

The Adapter Plugin, a core component of the DT framework, creates adapter-specific implementations of other DT components. This component can be created directly by the AWF or indirectly by Factory classes.

**Connection Module**

The connection component manages all properties necessary for the DT framework to communicate with the target application. These properties might include URLs, tenant identification, and authentication credentials. The connection component maintains a list of standard properties handled directly by the DT framework. Plugin implementations can add custom properties, which are handled separately. Plugins can also exclude certain properties.

Some connection properties are persisted in the Cloud Application Configuration file for use at runtime. Note that the Cloud Application Configuration file is a plain-text file and not appropriate for storing sensitive data or security credentials.

**Metadata Browser**

The Metadata Browser queries the target application for metadata such as APIs, operations, and business objects. The Metadata Browser constructs a Cloud Application Model (CAM) from the application’s metadata and caches the
information. Acquiring and storing metadata can involve several processes, including but not limited to:

- Processing source artifacts such as WSDLs, schemas, and WADLs. The DT framework has default classes for processing these artifacts and building a CAM. For example, the WSDLMetadataParser class can process a WSDL and produce a CAM where operations correspond to the operations in the WSDL and business objects (referred to as CloudDataObjects in the CAM) correspond to complex types in schemas referenced by the WSDL. Adapter plugins can add new parsers for formats not supported. In some cases, extending a parser can functionality.

- Calling a provider-specific metadata API. The DT framework only provides web-service clients for assisting in this process. Details of a provider’s metadata API, if it exists, are opaque to the DT framework. A metadata API might include business objects and operations that can be converted into a CAM. You can combine processing of artifacts such as WSDLs with metadata API artifacts. For example, an adapter might process a provider’s WSDL first to get the main business objects and operations, then process the provider’s metadata API to acquire additional information such as custom objects and permissions.

- Generating schemas from JSON samples. REST-based applications might have no source artifacts or metadata APIs that can be used to build a CAM. However, The DT framework provides a tool for generating an XML schema from a JSON sample: the oracle.cloud.connector.impl.rest.translator.RESTTranslator Java API under the runtime implementation JAR. This generated schema can then be used as a source artifact in the adapter.

The Models

The core DT framework model is the Cloud Application Model (CAM). The Metadata Browser fetches objects in the CAM. To generate the runtime metadata, the AWF constructs a transformation model (TM) and passes it to the Runtime Metadata Generation component. The TM contains mappings to the CAM. Each mapping represents a user-selected business object or operation and any overrides. An override might be changing the operation name or replacing a business object field. The TM is temporary because it only represents current selections by the wizard user. In contrast, the CAM represents the current state of the entire target application.

Runtime Metadata Generation

The Runtime Metadata Generation component accepts information from the DT framework and generates artifacts used by the RT framework. This component accepts the TM, which includes the Integration WSDL and Cloud Application Configuration file. Runtime metadata produced by this component can be changed when the adapter is in edit mode. For example, a wizard user might want to update the target operation being invoked.

Runtime Functional Components

This section describes the runtime functional component modules, including the runtime adapter shell, runtime adapter component, client invoker, payload tools, and runtime artifacts.

Runtime Adapter Shell

The Runtime Adapter Shell, a core component of the RT framework, is the entry point for runtime invocation. Service engines such as BPEL and Service Bus invoke the shell as a standard Java Connectivity Architecture (JCA) adapter. Upon invocation, the shell accepts the message payload, the Integration WSDL, and the Cloud Application
Configuration file. The shell invokes the Runtime Adapter Component, which processes the message and invokes the target application.

**Runtime Adapter Component**

For outbound invocation, the Runtime Adapter Component connects to and sends messages to the source or target application. This component translates the payload from the shell to a payload the target application can accept. This component also accepts and has access to the Integration WSDL, and the Cloud Application Configuration file. Finally, this component can use utility classes to convert messages from XML to JSON or the reverse when necessary.

For inbound invocation, the Runtime Adapter Component translates an application request payload to an integration-friendly request payload and translates an integration-friendly response payload to an application-specific response payload.

**Client Invoker**

To invoke remote operations, the Runtime Adapter Component uses a Client Invoker. Client Invokers are injected in a Runtime Plugins adapter class. There are two flavors of Client Invokers:

- SOAP Client Invokers, which send SOAP messages.
- REST Client Invokers, which send JSON/XML messages using HTTP protocol and the REST architectural style.

The client invokers do not need to be directly invoked. By default, messages are routed by client invokers after message transformation or session handling.

**Payload Tools**

The Runtime Adapter Component may need to perform payload processing. Payloads sent from the SOA or Service Bus engine are always initially in XML format. Similarly, payloads between the Runtime Adapter Component and the target application can be in various formats, such as SOAP or JSON. The RT framework provides tools that help with outbound payload processing, including:

- XML to JSON or JSON to XML conversion
- XPath queries
- WSDL parsing (JSR-110)

For inbound invocation, the payload sent by the target application to SOA or Service Bus is in the application’s native format, SOAP or JSON. The Runtime Adapter Component receives these messages and translates them to XML format before sending them to SOA or Service Bus for further processing.

**Runtime Artifacts**

This component consists of the classes and interfaces that process generated runtime artifacts such as the integration WSDL and Cloud Application Configuration file.

---

**Adapter Wizard Framework**

You use the Adapter Wizard Framework (AWF), also called the UI Framework, to configure adapter wizards at design time. Because this framework is generic, you can build once and deploy anywhere, across integration platforms such as Oracle SOA.
Suite, Oracle Service Bus, and Integration Cloud Service. The AWF consists of Core UI APIs and UI Objects.

**Core UI APIs**

Core UI APIs are interfaces and abstract classes that adapters implement or extend to provide content for the wizard pages.

**UI Objects**

The UI Objects framework provides EditFields (UI Fields) that you use to develop adapter-specific pages. EditFields are generic UI Components not tied to any UI technology. They are plain Java objects.

The following diagram shows the Adapter Wizard (UI) Framework topology:

![Diagram showing Adapter Wizard Framework](image)

**What the AWF Handles**

- Rendering wizard
- Collecting user inputs
- Handling wizard events and delegating them to the adapter
- Handling wizard navigations and parent/child page navigations
- Enabling/disabling of navigation buttons depending on which page (parent/child) is loaded
- Validating mandatory fields
- Invoking the Connector Architecture plugin

**What the Adapter Handles**

- Providing a list of pages with unique page IDs to AWF
- Providing content for each page (UI Fields)
• Validating business logic
• Adding/Removing of pages from pages list based on user actions/selections
• Adding/Removing of UI Fields from pages based on user actions/selections
• Generating and returning artifacts to the AWF

Adapter Development Life Cycle

The development life cycle of an Oracle Adapter includes these stages: development, packaging, registration, and consumption (or use).

The following diagram illustrates the four stages:

The four stages can be further summarized as follows:

1. **Adapter Development**: The adapter consists of two major components:
   - **Design-Time Classes**: This component is mainly driven by the wizard. You can customize and extend the wizard using the Cloud Application Wizard Development Framework (AWF). The wizard lets the wizard user graphically browse and select business objects and operations of interest for integration with SaaS or On-Premises applications.
   - **Runtime Classes**: This component performs endpoint invocation. It implements the Cloud Runtime SDK to interact with the enterprise service (SaaS or On-Premises) WSDL SOAP APIs. This component also performs tasks such as translation, session management, authentication, and authorization.

2. **Adapter Packaging**: Build the source code and package it as deployable (JAR) file.

3. **Adapter Registration**: Register the adapter with JDeveloper or ICS. Developers can self-publish their adapters.

4. **Adapter Consumption**: The client or SaaS application consumes the adapter.
Developing the Sample Cloud Adapter

This chapter describes the process of developing a cloud adapter named Sample Cloud Adapter based on a sample integration service. See the SampleCloudAdapter project for more detailed source code.

Topics

• Differences between Outbound and Inbound Adapters
• Setting Up the Cloud Adapter Development Environment
• Creating the Adapter Project
• Design-Time Components
• User Interface Module
• Plugin Module
• Runtime Components

Differences between Outbound and Inbound Adapters

Most of the steps for building outbound and inbound adapters are the same or similar. Major differences are listed in this section.

• The outbound wizard includes an Operations page, while the inbound wizard includes Request Configuration and Response Configuration pages instead. (Both include Welcome and Connection pages first and a Summary page last.)
• Only inbound adapters use an Endpoint WSDL or WADL file.
• Endpoint invocation is different for outbound and inbound adapters.
• Message transformation is different for outbound and inbound adapters.

In this chapter, steps for creating an outbound or inbound adapter are the same unless otherwise noted.

Setting Up the Cloud Adapter Development Environment

The Sample Cloud Adapter project (SampleCloudAdapterProj) has dependencies upon the following JAR files, which are required for successful compilation.

You can set up dependency libraries in the project properties to use these JAR files from the ORACLE_HOME location. Or you can extract these JAR files from ORACLE_HOME subdirectories. An example ORACLE_HOME location is C:\Oracle\Middleware\ORACLE_HOME.
<table>
<thead>
<tr>
<th>No</th>
<th>Name of the Library</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cloud-designtime-api.jar</td>
<td>ORACLE_HOME\soa\soa\modules \oracle.cloud.adapter_12.1.3</td>
</tr>
<tr>
<td>2</td>
<td>cloud-designtime-impl.jar</td>
<td>ORACLE_HOME\soa\soa\modules \oracle.cloud.adapter_12.1.3</td>
</tr>
<tr>
<td>3</td>
<td>cloud-runtime-api.jar</td>
<td>ORACLE_HOME\soa\soa\modules \oracle.cloud.adapter_12.1.3</td>
</tr>
<tr>
<td>4</td>
<td>cloud-runtime-impl.jar</td>
<td>ORACLE_HOME\soa\soa\modules \oracle.cloud.adapter_12.1.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unified UI Framework JARs</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Jdeveloper JARs</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Common JARs</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
</tbody>
</table>

**Creating the Adapter Project**

Follow these steps to create a project for the Sample Cloud Adapter. These steps are the same for an outbound or inbound adapter.

1. Open JDeveloper.

2. Create a new JDeveloper extension Application by selecting File > New > Application and typing Extension in the search field.
3. Select **Extension Application** from the list and click **OK**.

4. Enter an Application Name and Application Package Prefix.

5. Enter a Project Name and click **Next**.
6. Click Finish.

An empty project is displayed in the Project Explorer.

7. Add dependency libraries to the SampleCloudAdapter project. Right click on the SampleCloudAdapter project and select Project Properties.


9. Add the Cloud Adapter SDK and UI Object Framework JAR files from ORACLE_HOME locations (see Setting Up the Cloud Adapter Development Environment).
10. Click OK.

11. Double click on extension.xml and edit the values according to the new adapter.

- **Display Name** — Name of the cloud adapter
- **Identifier** — Unique identity of the `SampleCloudAdapter` plugin
- **Resource Bundle** — Adapter-specific resource bundle to tokenize the element property values

When you create an extension project, the extension manifest is created and opened in the overview editor. The adapter descriptor file is `extension.xml`. The order in which extensions are loaded depends on the entries in the extension manifest. The `MANIFEST.MF` file lists cloud adapter dependency JAR files.

12. Click the **Source** tab and edit the `trigger-hook` element in the `extension.xml` file as shown in the following example:

```
<trigger-hooks xmlns="http://xmlns.oracle.com/ide/extension">
  <triggers>
  </triggers>
</trigger-hooks>
```
Creating the Adapter Project

The sub-elements of the trigger-hook element are:

- **adapterType**@technology-keys — The type of the adapter depends on the integration platform. For example:
  
  **JDeveloper**: technology-keys="SOA,BPM"
  
  **Service Bus or ICS**: technology-keys="ServiceBusTechnology"

- **name** — Name of the adapter

- **category-name** — Name of a category for the new cloud adapter

- **description** — Description of the adapter

- **inbound** — Flag to enable inbound adapter capabilities

- **outbound** — Flag to enable outbound adapter capabilities

- **binding-type** — Binding type. For example:
  
  **JDeveloper**: "jca"
  
  **Service Bus or ICS**: "sb"

- **binding-subtype** — Binding subtype details. For example:
  
  **JDeveloper**: "jca"
  
  **Service Bus or ICS**: "jca/samplecloudadapter"

- **implementation-class** — Provider class and new adapter implementation class details
• icon16x16 and 20x20 — Adapter icon location details

13. Double click the MANIFEST.MF file and change the content as shown in the example below. This is required to include the Bundle-Name and runtime dependencies (Require-Bundle) in the new cloud adapter.

```
Bundle-ManifestVersion: 2.0
Bundle-Version: 12.1.2
Bundle-Name: oracle.cloud.adapter.sample
Bundle-SymbolicName: oracle.cloud.adapter.sample
Bundle-ClassPath: .
Require-Bundle:
oracle.external.oracle-jrf-webservices,
oracle.external.oracle-jrf-wsdl,
oracle.external.jaxws,
oracle.external.oracle-jrf-saaj,
oracle.sca.modeler,
oracle.sca.ui.adapters,
oracle.cloud.adapter.api,
oracle.cloud.adapter.impl,
oracle.javatools,
oracle.jewt-core,
oracle.external.balishare,
oracle.sb.tooling.ide.sca,
oracle.soa.tools.widgets,
oracle.bpm-ide-common,
oracle.uic,
oracle.ide,
oracle.tools.cloud.adapter.ide
```


15. Create this XML file under the src\META-INF directory of the plugin JAR file.
To create the content of the `cloud-adapter.xml` file, see Creating a cloud-adapter.xml File for a Cloud Adapter.

**Creating a cloud-adapter.xml File for a Cloud Adapter**

To enable adapter registration, an adapter plugin MUST contain an XML file named `cloud-adapter.xml` in the `META-INF` folder of the plugin JAR file.

The adapter registration service ONLY processes files named `cloud-adapter.xml` and ONLY looks for these files under `META-INF`.

Creating a `cloud-adapter.xml` file is one step in the overall task Creating the Adapter Project. Use this section and the following table to create the content of this file.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>(Optional) A description of this ApplicationType. Should not describe the adapter but the actual SaaS application. Currently this element is not cannot be localized.</td>
</tr>
<tr>
<td>displayName</td>
<td>(Optional) The user-friendly format of the SaaS application. This should not be the display name of the adapter but the actual SaaS application name.</td>
</tr>
<tr>
<td>adapterPluginID</td>
<td>(Required) A simple, unique name for the adapter. This name MUST be <strong>unique</strong> across all adapters. This is used as a reference for associating the adapters with an ApplicationType.</td>
</tr>
<tr>
<td>UIProviderClass</td>
<td>(Optional) If the adapter is registered with ICS, then this element must specify the fully qualified name of the class that implements CloudAdapterProvider from the UI Framework.</td>
</tr>
<tr>
<td>Element</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>messageExchangePatterns</td>
<td>(Required) A list of message exchange patterns supported by this plugin. Refer to this API: oracle.tip.tools.ide.adapters.cloud.api.mod el.InvocationStyle. Samples: REQUEST_RESPONSE, ONEWAY, ASYNCHRONOUS, NOTIFICATION, PUBSUB, CALLBACK;</td>
</tr>
<tr>
<td>defaultSecurityPolicy</td>
<td>(Required) Specifies which security policy to use by default.</td>
</tr>
<tr>
<td>managedSecurity</td>
<td>(Optional) Set managedSecurity to true if security is handled by the platform (for example, by OWSM). Set it to false if the adapter handles its own security.</td>
</tr>
<tr>
<td>propertyDefinitions</td>
<td>(Required) A list of propertyDefinition elements which define adapter connection properties. Refer to these APIs: • oracle.tip.tools.ide.adapters.cloud.api. connection.PropertyDefinitions • oracle.tip.tools.ide.adapters.cloud.api. connection.PropertyGroup • oracle.tip.tools.ide.adapters.cloud.api. connection.PropertyType Samples: STRING, NUMBER, URL, WSDL_URL, XSD_URL, WADL_URL, FILE, PASSWORD, URL_OR_FILE</td>
</tr>
<tr>
<td>adapterFactory</td>
<td>(Required) ALL adapters MUST provide the classname of a class that implements the oracle.tip.tools.ide.adapters.cloud.api.plugin.CloudApplicationAdapterFactory interface.</td>
</tr>
<tr>
<td>ResourceBundle</td>
<td>(Optional) A resource bundle for NLS. You can specify a fully qualified class name or a resource path.</td>
</tr>
<tr>
<td>shortDescription</td>
<td>(Optional) A brief description of the adapter.</td>
</tr>
<tr>
<td>pluginVersion</td>
<td>(Required) The version of the adapter.</td>
</tr>
<tr>
<td>Element</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Features</td>
<td>(Optional) A list of features this adapter requires.</td>
</tr>
<tr>
<td>vendorInfo</td>
<td>(Optional) A consolidation of vendor metadata.</td>
</tr>
<tr>
<td>supportedEndpointsLocation</td>
<td>(Optional) An enumeration of values that determine where this adapter is being registered, for example, onPremise or onCloud.</td>
</tr>
<tr>
<td>sdkVersion</td>
<td>(Required) The Cloud Adapter SDK version on which this adapter was built.</td>
</tr>
</tbody>
</table>

The following is an example `cloud-adapter.xml` file.

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<tns:AdapterPluginConfig xmlns:tns="http://xmlns.oracle.com/adapters/cloud">
  <adapterPluginID>sampleadapter</adapterPluginID>
  <displayName>Oracle Sample Cloud Adapter</displayName>
  <description>Oracle Sample Cloud Adapter</description>
  <adapterFactory>{APPLICATION FACTORY CLASS}</adapterFactory>
  <supportedSecurityPolicies>
    <policy>USERNAME_PASSWORD_TOKEN</policy>
  </supportedSecurityPolicies>
  <defaultSecurityPolicy>USERNAME_PASSWORD_TOKEN</defaultSecurityPolicy>
  <propertyDefinitions>
    <propertyDefinition>
      <propertyName>targetWSDLURL</propertyName>
      <propertyType>WSDL_URL</propertyType>
      <propertyGroup>CONNECTION_PROPS</propertyGroup>
      <required>true</required>
      <persistent>true</persistent>
    </propertyDefinition>
    <displayName>WSDL URL</displayName>
  </propertyDefinitions>
  <UIProviderClass>oracle.tip.tools.ide.adapters.samplecloudadapter.wizardSAMPLECLOUDADAPTERPROVIDER</UIProviderClass>
  <sdkVersion>2.1.0</sdkVersion>
</tns:AdapterPluginConfig>
```

The following diagram shows the importance of the adapterFactory and UIProviderClass elements in the `cloud-adapter.xml` file.
Design-Time Components

The design-time components consist of the User Interface Module and the Plugin Module. These components create the wizard. The runtime components use the data accepted by the wizard to connect to the target application and perform selected operations or request/response tasks.

You build the design-time components using the Adapter SDK APIs listed in the following table.

<table>
<thead>
<tr>
<th>Wizard Step</th>
<th>User Interface Module Actions</th>
<th>Plugin Module Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Wizard Initialization</td>
<td>Wizard is initialized based on values defined in <code>extension.xml</code> or <code>cloud-adapter.xml</code>. Examples: JDeveloper — <code>extension.xml</code>, <code>param: implementation-class</code> ICS or SB — <code>cloud-adapter.xml</code>, <code>param: UIProviderClass</code></td>
<td>None</td>
</tr>
</tbody>
</table>
### 2. Connection page

1. Create a new instance of the plugin.
2. Get `CloudConnection` interface from the plugin.
3. Request user input for connection properties (such as username, password, WSDL URL, and so on).
4. Invoke the `ping()` function on the `CloudConnection` interface (from the plugin).

Note: You don’t have to write a completely new connection page for each adapter. You can use `CloudConnectionPage` from the SDK, and page controls can be derived from the `cloud-adapter.xml` file.

### 3. Metadata browsing page

1. Get an instance of `CloudMetadataBrowser` from the plugin.
2. Invoke the `loadMetadata()` function on the `CloudMetadataBrowser` instance.
3. Display a dropdown of APIs from the `CloudMetadataBrowser` instance.
4. Display a list of operations from the `CloudMetadataBrowser` instance based on the selected API.
5. Display a list of business objects from the `CloudMetadataBrowser` instance based on the selected operation.
6. Facilitate the setting of header and processing options based on the selected operation from the `CloudMetadataBrowser` instance.

### 4. Query support (optional)

1. If the user selected a Query operation in #3, provide a UI for user to enter a query string, plus contextual help.
2. Provide a test button for the wizard user to test a query.
3. Provide a UI for the wizard user to enter values for bind variables.

### Design-Time Components

2-12 Developing Custom Oracle Cloud Adapters
5. Artifact generation

i) Create an instance of the Transformation Model, which contains all selected objects and changes ("transformations") that need to take place during generation.

ii) Create file system abstractions for where generated files need to be written.

iii) Get an instance of RuntimeMetadataGenerator from the plugin.

iv) Set the TransformationModel on the RuntimeMetadataGenerator instance and invoke the generate() method.

i) Return an instance of RuntimeMetadataGenerator.

ii) Generate artifacts when the generate() function of RuntimeMetadataGenerator is invoked. There are built-in artifact generators for the integration WSDL and cloud application configuration (CAC) file. Any additional files can also be generated here if necessary.

User Interface Module

The User Interface Module centers on the classes that define the pages in the wizard.

The following diagram shows the high-level design of Framework classes and subclasses that adapters must implement. All classes starting with name Sample* are specific to the Sample Cloud Adapter and used as examples throughout this chapter. Each adapter must implement its wizard in a similar way.

![Diagram showing the high-level design of Framework classes and subclasses.]

Note: The current adapter implementation is specific to the JDeveloper plugin.
**Adapter User Interface Entry Point**

Create the entry point for the adapter plugin.

Refer to an implementation class called `SampleCloudScaEndpointImpl` as specified in `extension.xml`.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.samplecloudadapter.wizard</code></td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td>The SCAEndpoint is invoked when a new adapter is dragged and dropped into the SOA diagram or when an existing adapter is double-clicked.</td>
</tr>
<tr>
<td><code>SampleCloudScaEndpointImpl</code></td>
<td></td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td>Initiates the Sample Cloud Adapter Wizard. References the UIBinding class (see the class diagram).</td>
</tr>
<tr>
<td><code>runCreateWizard()</code></td>
<td></td>
</tr>
</tbody>
</table>
Implement the `runCreateWizard` method. This method references the `UIBinding` class.

```
ICloudAdapterUIBinding iCloudAdapterUIBinding = new SampleCloudUIBinding(uiHandler.getFilter(), uiHandler.getLocale(), adapterId);
```

**Note:** You must implement the UI Provider Class to register the adapter with ICS. The code snippet below is in `SampleCloudAdapterProvider`. This class references the `UIBinding` class as specified in diagram.
public class SampleCloudAdapterProvider extends CloudAdapterUIProvider
{
    public ICloudAdapterUIBinding getCloudAdapterUIBinding(CloudAdapterFilter cloudAdapterFilter, Locale locale)
        throws CloudAdapterException
    {
        SampleCloudUIBinding uiBinding = null;
        try
        {
            uiBinding = new SampleCloudUIBinding(cloudAdapterFilter, locale);
        } catch (Exception ex) {
            throw new CloudAdapterException(ex.getMessage());
        }
        return uiBinding;
    }

    @Override
    public String getLocalizedAdapterType(Locale locale) {
        return "SampleCloudAdapterLab";
    }
}

Here is the corresponding cloud-adapter.xml entry:

<UIProviderClass>
oracle.tip.tools.ide.adapters.samplecloudadapter.wizard.SampleCloudAdapterProvider</UIProviderClass>

Refer to a UI binding class called SampleCloudAdapterUIBinding.

<table>
<thead>
<tr>
<th>Package, Class and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.wizard</td>
</tr>
</tbody>
</table>

<p>| <strong>Class:</strong> | SampleCloudAdapterUIBinding |
| <strong>The implementation class of ICloudAdapterUIBinding:</strong> |
| - Creates the DefaultAdapterPluginContext. This is an adapter-specific context for creating artifacts. This context object is passed to all pages, so pages can pass values from one page to other. |
| - Creates the actual adapterProvider (subclass of AbstractCloudApplicationAdapter). Sets this in the context created in the previous step. |
| - Has APIs to get the list of pages, get the fields for each page, perform business logic validation, delegate page actions to respective pages, and generate artifacts. Handles interactions between the wizard and the actual adapter pages. |</p>
<table>
<thead>
<tr>
<th>Method:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getEditPages()</td>
<td>Returns the list of pages to display in the wizard. The key in the map is the pageId and the value is an instance of ICloudAdapterPage. The page sequence in the map determines the page sequence in the wizard.</td>
</tr>
<tr>
<td>generateMetadataArtifacts()</td>
<td>Returns CloudAdapter runtime artifacts such as the Cloud Adapter Configuration file and the integration WSDL based on user selections.</td>
</tr>
</tbody>
</table>
As specified in diagram, add the wizard pages into `getEditPages()` to display the page sequence.

- Welcome Page: For Welcome Screen
- Connection Page: To provide connectivity details
• Operations Page: To choose cloud operation and respective business objects (outbound only)

• Request Configuration Page: To choose request type business object (inbound only)

• Response Configuration Page: To choose type of integration and to choose the reply type business object (inbound only)

• Summary Page: To display configured information

```java
public LinkedHashMap<String, ICloudAdapterPage> getEditPages(Object adapterConfiguration) {
    LinkedHashMap<String, ICloudAdapterPage> editPages = new LinkedHashMap<String, ICloudAdapterPage>();
    String referenceName = context.getReferenceBindingName();
    editPages.put(CloudAdapterConstants.WELCOME_PAGE_ID, new SampleCloudAdapterWelcomePage(context));

    // Skip connection page if applicationInstanceId is present.
    CloudAdapterFilter filter = (CloudAdapterFilter)context.getContextObject(CloudAdapterConstants.UI_CLOUD_ADAPTER_FILTER);
    if(filter.isAddConnection()) {
        editPages.put(CloudAdapterConstants.CONNECTION_PAGE_ID, new CloudAdapterConnectionPage(context));
    } else if (!filter.isInbound()) {
        editPages.put(CloudAdapterConstants.OPERATIONS_PAGE_ID, new SampleCloudAdapterOperationsPage(context));
    } else if (filter.isInbound()) {
        editPages.put(SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REQ_CONFIG_PAGE_ID, new SampleInboundRequestConfigPage(context));
        editPages.put(SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REP_CONFIG_PAGE_ID, new SampleInboundReplyConfigPage(context));
    }
    editPages.put(CloudAdapterConstants.SUMMARY_PAGE_ID, new SampleCloudAdapterSummaryPage(context));

    return editPages;
}
```

Implementation of the `generateMetadataArtifacts()` method is optional. It is required if you wish to add any additional parameters to the Cloud Application Configuration (CAC) file.

```xml
<connection-factory location="cloud/CloudAdapter">
    <non-managed-connection
        managedConnectionFactoryClassName="oracle.tip.tools.ide.adapters.samplecloudadapter.runtime.SampleCloudAppConnectionFactory">
        <property name="targetWSDLURL" value="WSDLs/"/>
```
Refer to the SampleCloudAdapter class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.plugin</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleCloudAdapter</td>
<td>Creates and initializes the application-specific adapter.</td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td></td>
</tr>
<tr>
<td>getConnection()</td>
<td>Returns the adapter’s Connection object.</td>
</tr>
<tr>
<td>getName()</td>
<td>Returns the name of the adapter.</td>
</tr>
</tbody>
</table>

Implement these methods:

- `getConnection()` — To get the cloud connection object
- `getName()` — To get the name of the adapter

**Note:** The remaining methods (`getMetadataBrowser` and `getRuntimeMetadataGenerator`) are implemented after the first check.

```java
// Get adapter connection object
public CloudConnection getConnection() {  
    return new SampleCloudAdapterConnection(getPluginContext());  
}

@Override
public CloudMetadataBrowser getMetadataBrowser(CloudConnection connection) {  
    return null;  
}

@Override
public RuntimeMetadataGenerator getRuntimeMetadataGenerator() {  
    return null;  
}

@Override
public String getName() {  
    return "Sample Cloud Adapter";  
}
```

**Welcome Page**

Create the Welcome page in the wizard.

Refer to the SampleCloudAdapterWelcomePage class.
<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.wizard</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleCloudAdapterWelcomePage</td>
<td>This class builds an adapter-specific Welcome page.</td>
</tr>
<tr>
<td>getWelcomeText()</td>
<td>Returns welcome text, which briefly describes the page.</td>
</tr>
</tbody>
</table>

This class can inherit following API to modify the default WelcomePage properties. Here is a code snippet that shows how to extend CloudAdapterWelcomePage and change the Welcome Page text details.

```java
import oracle.tip.tools.adapters.cloud.impl.CloudAdapterWelcomePage;

public class SampleCloudAdapterWelcomePage extends CloudAdapterWelcomePage {
    public String getWelcomeText() {
```
return
SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.WELCOME_PAGE_WELCOME_TEXT);
}

Refer to the SampleCloudAdapterCloudText util class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.util</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleCloudAdapterCloudText</td>
<td>This adapter-level Util class fetches wizard-specific text from the resource bundle, such as welcome text, page names, and so on.</td>
</tr>
<tr>
<td>getBundle()</td>
<td>Loads the resource bundle.</td>
</tr>
<tr>
<td>getString()</td>
<td>Gets text based on the CONSTANT variable.</td>
</tr>
</tbody>
</table>

For configuration purposes, using a properties file is good for reuse. In this way, when the code is packaged in a JAR file, other users can put different configurations in the SampleCloudAdapterBundle.properties file. The following code snippet reads the properties file.

The getBundle() and getString() methods load and fetch resource bundle information.
import java.util.ResourceBundle;

public class SampleCloudAdapterCloudText {

    //Create resource bundle (SampleCloudAdapterBundle.properties) file and specify with package details
    public static final String SAMPLE_CLOUD_CONNECTOR_BUNDLE =
            "oracle.tip.tools.ide.adapters.samplecloudadapter.util.SampleCloudAdapterBundle";

    //To get resource bundle
    public static ResourceBundle getBundle() {
        return ResourceBundle.getBundle(SAMPLE_CLOUD_CONNECTOR_BUNDLE);
    }

    public static String getString(String key) {
        if(getBundle() == null)
            return key;
        return getBundle().getString(key);
    }

    // Welcome page text description will be available in resource bundle
    public static final String WELCOME_PAGE_WELCOME_TEXT =
            "samplecloudadapter.welcome.page.welcome.text";
}

Create the properties file in the util package to define the wizard UI text information in key-value pair mode.

Right click on oracle.tip.tools.ide.adapters.samplecloudadapter.util and Select New > From Gallery. Type File in the search field and select File (General). Then enter the Name as SampleCloudAdapter.Properties.

Here is content from the Resource bundle. This content includes all the wizard pages of text information.

samplecloudadapter.adapter.type=samplecloudadapter

# Welcome page
samplecloudadapter.welcome.page.welcome.text=This wizard helps you create a service using samplecloudadapter Cloud Adapter. You will be asked to specify configuration parameters and define an operation for the service.

**Connection Page**

Create the Connection page in the wizard.

You can leverage the UI Framework default connection page called CloudAdapterConnectionPage. A default connection page is loaded and constructed based on the values defined in cloud-adaper.xml.

<propertyDefinitions>
    <propertyDefinition>
        <propertyName>targetWSDLURL</propertyName>
        <propertyType>WSDL_URL</propertyType>
        <propertyGroup>CONNECTION_PROPS</propertyGroup>
        <required>true</required>
        <persistent>true</persistent>
        <displayName>WSDL URL</displayName>
    </propertyDefinition>
</propertyDefinitions>
Refer to the `SampleCloudAdapterConstants` class to declare adapter-level constants.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package:</td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.util</td>
<td></td>
</tr>
<tr>
<td>Class:</td>
<td></td>
</tr>
<tr>
<td>SampleCloudAdapterConstants</td>
<td>Declares adapter-level constant variables and values.</td>
</tr>
</tbody>
</table>

Declare the adapter-level constants. Ensure that the following variables are defined for the new custom adapter.

```java
// Define Custom namespace for integration WSDL (runtime artifact). Otherwise, use SaaS application target namespace
public static final String ENDPOINT_WSDL_DEFAULT_TARGET_NS = "http://xmlns.oracle.com/pcbpel/cloud/connector/samplecloudadapter";

// SaaS Application Target namespace
public static final String SAMPLE_SERVICE_NAMESPACE="http://xmlns.oracle.com/apps/crmCommon/salesParties/accountService/";

// SaaS Application Message / Schema namespace

// Provision to configure service catalogue method name or configure any service operation name to test the connectivity to SaaS application (Ping method)
public static final String SAMPLE_CATALOG_SERVICE_OP_NAME="getAccount";

//Runtime Adapter Connection factory class name (outbound only)
public static final String CONNECTION_FACTORY_CLASS = "oracle.tip.tools.ide.adapters.samplecloudadapter.runtime.SampleCloudAppConnectionFactory";
```
If you would like to verify the Welcome and Connection Pages in JDeveloper before creating other wizard pages, you can follow these steps:

1. Refer to and implement the `SampleCloudAdapterConnection` class. See Connection Properties and Testing.

2. Build the Sample Cloud Adapter. See Building the Sample Cloud Adapter.

3. Package the Sample Cloud Adapter. See Packaging and Generating the JAR File.

4. Register the adapter with JDeveloper. See Registering the Adapter.

5. Restart the WebLogic SOA Server.

### Operations Page for Outbound Adapters

For an outbound adapter, create the Operations page in the wizard. Refer to the `SampleCloudAdapterOperationsPage` class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.samplecloudadapter.wizard</code></td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td><code>SampleCloudAdapterOperationsPage</code></td>
<td>This class implements the adapter’s Operations page, which displays application-specific business operations and corresponding data objects.</td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td></td>
</tr>
<tr>
<td><code>getPageId()</code></td>
<td>Gets the page ID and initializes class-level variables.</td>
</tr>
<tr>
<td><code>getPageName()</code></td>
<td>Gets the page name.</td>
</tr>
<tr>
<td><code>getWelcomeText()</code></td>
<td>Gets the page’s welcome text.</td>
</tr>
<tr>
<td><code>getPageTitle()</code></td>
<td>Gets the page title.</td>
</tr>
<tr>
<td><code>getPageEditFields()</code></td>
<td>Returns the page fields.</td>
</tr>
<tr>
<td><code>getUpdatedEditPages()</code></td>
<td>Updates the page controls based on user selections.</td>
</tr>
<tr>
<td><code>updateBackEndModel()</code></td>
<td>Updates the back-end data model.</td>
</tr>
</tbody>
</table>
The Sample Cloud Adapter extends the default Operations page, modifies UI objects, and groups all the operations into a single API. To display the API, Operations, and data objects:

```java
// Get metadatabrowser for our application
CloudMetadataBrowser browser = CloudAdapterUtils.getMetadataBrowser(adapterPluginContext);

// To get the list of API - browser is to
List<CloudAPINode> apis = browser.getAPIs();

// To display API Label and Description
createOrUpdateField(editFields, CloudAdapterConstants.CLOUD_API,
    CloudAdapterText.getString(locale, CloudAdapterText.OPERATIONS_PAGE_CLOUD_API_LABEL),
    CloudAdapterText.getString(locale, CloudAdapterText.OPERATIONS_PAGE_CLOUD_API_DESC),
    false, apisObj, EditField.LabelFieldLayout.ONE_ROW_LAYOUT, CloudAdapterText.getString(locale,
    CloudAdapterText.OPERATIONS_PAGE_CLOUD_API_INFO_TEXT),
    3, EditField.LabelFieldAlignment.LEFT_LEFT);

UIObject apiVersionObj = UIFactory.createTextBox(apiNode.getVersion(),
    TextBoxObject.DEFAULT_SIZE, true);

To display Operations and businessObjects:

// To get list of operations based on API
List<CloudOperationNode> operationNodes = browser.getOperations(firstApiNode);

// Display list operations
UIObject operationsObj = UIFactory.createSelectObject(operationNames,
    operationNames, targetOp,
    ISelectObject.DISPLAY_LIST, Boolean.TRUE);
```
To get the list of business objects:

```java
List<CloudDataObjectNode> dataObjects = browser.getDataObjectNodes(operationNode);
```

### Request Configuration Page for Inbound Adapters

For an inbound adapter, create the Request Configuration page in the wizard. Refer to the `SampleInboundRequestConfigPage` class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.wizard</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleInboundRequestConfigPage</td>
<td>This class implements the Request Configuration page.</td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td></td>
</tr>
<tr>
<td>getPageName()</td>
<td>Gets the page name.</td>
</tr>
<tr>
<td>getPageTitle()</td>
<td>Gets the page title.</td>
</tr>
<tr>
<td>getWelcomeText()</td>
<td>Gets the page’s welcome text.</td>
</tr>
<tr>
<td>getPageEditFields()</td>
<td>Returns the page fields.</td>
</tr>
<tr>
<td>Populate()</td>
<td>Populates the page with business objects and business object descriptions.</td>
</tr>
</tbody>
</table>
Implement the `iCloudAdapterPage` for `SampleCloudRequestConfigPage`, because the UI Framework does not have any abstract implementation.

Implement the `getPageEditFields()` methods to add page-level fields and populate respective values into the request configuration page.

```java
// To get Metadata browser object
CloudMetadataBrowser browser = CloudAdapterUtils.getMetadataBrowser(adapterPluginContext);
// populate required controls in request configuration page.
populate(currentPageFields, browser, null);
```

Implement the supporting private method called `populate()` to load and display service-level business objects.

```java
//UI controls to display list of business objects of respective SaaS application service.
```
Override the `updateBackEndModel()` method to maintain the state of the request configuration page. The back-end transformation model generates runtime artifacts such as the JCA and WSDL based on wizard user selections.

To get the state of the Adapter Page

```java
CloudAdapterPageState state = new CloudAdapterPageState(false, wizardPages, currentPageFields);
//To transfer state details to backend model to construct runtime artifacts.
transferUIDataToModel(props, map);
```

### Response Configuration Page for Inbound Adapters

For an inbound adapter, create the Response Configuration page in the wizard.

Refer to the `SampleInboundReplyConfigPage` class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.samplecloudadapter.wizard</code></td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td>This class builds the Response Configuration page.</td>
</tr>
<tr>
<td><code>SampleInboundReplyConfigPage</code></td>
<td></td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td>Returns a list of default fields to be displayed on the page.</td>
</tr>
<tr>
<td><code>getPageEditFields()</code></td>
<td></td>
</tr>
<tr>
<td><code>getPageName()</code></td>
<td>Gets the page name.</td>
</tr>
<tr>
<td><code>getPageTitle()</code></td>
<td>Gets the page title.</td>
</tr>
<tr>
<td><code>getHelpId()</code></td>
<td>Gets the page help ID.</td>
</tr>
</tbody>
</table>
Implement the `iCloudAdapterPage` for `SampleCloudRequestConfigPage`, because the UI Framework does not have any abstract implementation.

Implement the private supportive methods called `populate()` and `populateBusinessObjects()`. These load response business objects and data objects.

```java
// send check box
CheckBoxObject sendCheckBox = UIFactory.createCheckbox(SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REP_CONFIG_PAGE_SEND_RES_CHECKBOX, true, true);

SampleCloudAdapterUtil.createOrUpdateField(currentPageFields, SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REP_CONFIG_PAGE_SEND_RES_CHECKBOX, x,
```
SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.SAMPLE_CLOUD_APP_INBOUND_REP_CONFIG_PAGE_SEND_RES_CHECKBOX_LABEL), null, false, sendCheckBox, EditField.LabelFieldLayout.ONE_ROW_LAYOUT, "", 0, null);

// immediate/delayed response list box
String[] resTypeValues = {SampleCloudAdapterConstants.IMMEDIATE_RESPONSE};
String[] resTypeFormattedValues = {SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.SAMPLE_CLOUD_APP_INBOUND_REP_CONFIG_PAGE_IMMEDIATE_RESPONSE_VALUE)};

SelectObject resTypeSelectObject = UIFactory.createSelectObject(resTypeValues, resTypeFormattedValues,
SampleCloudAdapterConstants.IMMEDIATE_RESPONSE, ISelectObject.DISPLAY_LIST, true);

SampleCloudAdapterUtil.createOrUpdateField(currentPageFields,
SampleCloudAdapterConstants.RESPONSE_TYPE_LISTBOX,
SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.SAMPLE_CLOUD_APP_INBOUND_REP_CONFIG_PAGE_RESPONSE_TYPE_LISTBOX_LABEL), "", false, resTypeSelectObject, EditField.LabelFieldLayout.ONE_ROW_LAYOUT, null, 0, null);

// send fault check box
CheckBoxObject sendFaultCheckBox = UIFactory.createCheckbox(SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REP_CONFIG_PAGE_SEND_FAULT_CHECKBOX, true, true);

SampleCloudAdapterUtil.createOrUpdateField(currentPageFields,
SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REP_CONFIG_PAGE_SEND_FAULT_CHECKBOX,
SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.SAMPLE_CLOUD_APP_INBOUND_REP_CONFIG_PAGE_SEND_FAULT_CHECKBOX_LABEL), "", false, sendFaultCheckBox, EditField.LabelFieldLayout.ONE_ROW_LAYOUT, null, 0, null);

// separator
ISeparatorObject separator = UIFactory.createSeparatorObject();
SampleCloudAdapterUtil.createOrUpdateField(currentPageFields,
SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REQ_CONFIG_PAGE_SEPARATOR,
"", "", separator);

// create list box
if (selectedBizObj == null) {
SelectObject selectBizObject = UIFactory.createSelectObject(this.bizObjArray, this.bizObjArray, bizObjArray[0], ISelectObject.DISPLAY_LIST_AS_LISTBOX, false, true,
SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.SAMPLE_CLOUD_APP_INBOUND_REQ_CONFIG_PAGE_BUSINESS_OBJ_LIST_FIELD),
SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.SAMPLE_CLOUD_APP_INBOUND_REQ_CONFIG_PAGE_BUSINESS_OBJ_LIST_LABEL));
SampleCloudAdapterUtil.createOrUpdateField(currentPageFields,
SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REQ_CONFIG_PAGE_BUSINESS_OBJ_LIST_FIELD,
SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.SAMPLE_CLOUD_APP_INBOUND_REQ_CONFIG_PAGE_BUSINESS_OBJ_LIST_LABEL),
SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.SAMPLE_CLOUD_APP_INBOUND_REQ_CONFIG_PAGE_BUSINESS_OBJ_LIST_FILTER_PLACEHOLDER_TEXT));
}
Override the `updateBackEndModel()` method to transfer the response configuration information to the back-end model or cache.

```java
CloudAdapterPageState state = new CloudAdapterPageState(false, wizardPages, currentPageFields);
transferUIDataToModel(props, map);
```

Set `OperationMapping` for the Artifact Generator. As discussed previously, the DT framework generates the following files and artifacts:

- Cloud Application Configuration File (JCA)
- Integration WSDL
- SOAP WSDL or REST WADL

```java
TransformationModelBuilder integrationMB =
    SampleCloudAdapterUtil.createTransformationModelBuilder(adapterPluginContext, SampleCloudAdapterConstants.ARTIFACT_KEY_REQUEST);
```

Summary Page

Create the Summary page in the wizard.

Refer to the `SampleCloudAdapterSummaryPage` class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.wizard</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleCloudAdapterSummaryPage</td>
<td>This class builds the Summary page, which summarizes the adapter configuration information.</td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td></td>
</tr>
<tr>
<td>getPageEditFields()</td>
<td>Returns a list of fields to be displayed on the page.</td>
</tr>
<tr>
<td>getWelcomeText()</td>
<td>Gets the page’s welcome text.</td>
</tr>
</tbody>
</table>
setupOutboundSummary() Displays outbound target operation information on the summary page.

For an outbound adapter:

For an inbound adapter:
The `SampleCloudAdapter` class inherits and extends the UI Framework’s default Summary page properties. It overrides `getWelcomeText()`.

```java
public String getWelcomeText() {
    return SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.SUMMARY_PAGE_WELCOME_TEXT);
}
```

For an outbound adapter, implement the `setupOutboundSummary()` method to display the selected and configured information on the Summary page.

```java
private void setupOutboundSummary(List<EditField> headerFields) {
    this.displayText(headerFields, (String)
        adaPluginContext.getContextObject(CloudAdapterConstants.CLOUD_BIZ_OBJ),
        CloudAdapterConstants.CLOUD_BIZ_OBJ, "Business Object:","Business Object");

    this.displayText(headerFields, (String)
        adaPluginContext.getContextObject(CloudAdapterConstants.CLOUD_OPERATION),
        CloudAdapterConstants.CLOUD_OPERATION, "Operation Name:","Operation Name");
}
```

```java
private void displayText(List<EditField> headerFields, String text, String fieldName, String label, String desc){
    TextBoxObject uiObj = UIFactory.createTextBox(text, true);
    headerFields.add(UIFactory.createEditField(fieldName, label, desc, uiObj));
}
```
For an inbound adapter, implement supporting methods such as `setupInboundSummary()` to construct the back-end data model and display request and response information for the selected data objects.

```java
private void setupInboundSummary(List<EditField> headerFields)
{
    this.addSeparator(headerFields);
    this.displayText(headerFields, (String) adaPluginContext.getContextObject(CloudAdapterConstants.CLOUD_BIZ_OBJ),
                     CloudAdapterConstants.CLOUD_BIZ_OBJ,"Inbound Request Object:`","Inbound Request Object");
    this.addSeparator(headerFields);
    this.displayText(headerFields, (String) adaPluginContext.getContextObject(CloudAdapterConstants.CLOUD_BIZ_OBJ),
                     CloudAdapterConstants.CLOUD_BIZ_OBJ,"Inbound Response Object:`","Inbound Response Object");
}
```
Plugin Module

The plugin module connects to the source or target application, parses the metadata information, and generates the runtime artifacts (JCA and WSDL).

The following diagram shows the plugin classes in the Sample Cloud Adapter.

Connection Properties and Testing

The CloudConnection class stores and tests the connection properties used in the Connection page of the wizard.

The abstract class AbstractCloudConnection provides default implementations of methods for retrieving and storing connection properties. You must implement the ping() method. At a minimum, this method must validate the current set of properties in the connection object; see getProperties(), and setProperties(properties). The ping() method typically calls the application provider using all the information a user has provided. Upon a successful response, the method returns an instance of PingStatus with the success flag set to true, otherwise the success flag should be false and information regarding the failure should be provided using the PingStatus.getFailureMessage() method. No exception should be thrown from this method.

The following sequence diagram shows how this plugin instance is acquired.
As the diagram shows, the wizard first looks for a plugin instance by calling `getAdapter()` on the `oracle.cloud.connector.CloudAdapterFactory` class at runtime. The `getAdapter()` method is explained in later sections. Each plugin must provide a unique ID that is passed to the `getAdapter()` call and always returns an instance (new on first call) of `oracle.cloud.connector.CloudAdapter`. Once the wizard has a plugin’s implementation of `CloudAdapter`, it acquires the plugin’s implementation of the interface `oracle.cloud.connector.connection.CloudConnection`. An instance of `CloudConnection` acquires properties for connecting and authenticating to a specific cloud application.

Refer to the `SampleCloudAdapterConnection` class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.samplecloudadapter.plugin</code></td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td><code>SampleCloudAdapterConnection</code></td>
<td>This class tests SaaS application endpoint connectivity. It stores connection and authentication properties.</td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td></td>
</tr>
<tr>
<td><code>ping()</code></td>
<td>Tests endpoint connectivity.</td>
</tr>
<tr>
<td><code>createRequestPingMessage()</code></td>
<td>Creates a new SOAP request based on the operation being tested.</td>
</tr>
<tr>
<td><code>getPingStatusFromResponse()</code></td>
<td>Gets the status from the ping response.</td>
</tr>
<tr>
<td><code>getAuthenticationScheme()</code></td>
<td>Stores security credentials.</td>
</tr>
</tbody>
</table>
Override the ping() method to test endpoint connectivity. This is a custom implementation, where you can add a SOAPMessage and send a sample request to the operation defined in the Constants class.

For an outbound adapter:

```java
// SampleCloudAdapterConstants
public static final String SAMPLE_CATALOG_SERVICE_OP_NAME="getAccount";

// Gets the WSDL URL
String wsdlURL = getConnectionProperties().getProperty(AdapterConstants.SOURCE_WSDL_LOCATION);

// To create a request message
SOAPMessage message = this.createRequestPingMessage(soapHelper);

// Send SOAP request
SOAPMessage response = soapHelper.sendSOAP(message,getConnectionProperties(), endpointURL);

// createRequestPingMessage is adding operationname and namespace to SOAP message body
body.addChildElement(new QName(
    SampleCloudAdapterConstants.SAMPLE__MESSAGES_NAMESPACE,
    SampleCloudAdapterConstants.SAMPLE_CATALOG_SERVICE_OP_NAME));
```

For an inbound adapter:

```java
// Gets the WSDL URL
String wsdlURL = getConnectionProperties().getProperty(AdapterConstants.SOURCE_WSDL_LOCATION);
```
// Create SOAP Request
SOAPMessage message = this.createRequestPingMessage(soapHelper);

// Send SOAP request to target
SOAPMessage response = soapHelper.sendSOAP(message, getConnectionProperties(), endpointURL);

### Metadata Browser

The Metadata Browser retrieves and parses metadata from the source or target application so the wizard user can choose business objects.

The `CloudMetadataBrowser` interface retrieves and displays application metadata: operations, objects, faults, and so on. Instead of implementing the entire `CloudMetadataBrowser` interface, you can extend the abstract class `AbstractMetadataBrowser`. This class has some initialization code and built-in features such as parsing the service interface or schema and building the Cloud Application Model (CAM). The CAM consists of APIs, Operations, Objects, and Faults (if any). You only need to implement the few methods in this class that are pure abstract methods, while many tasks are already handled by the abstract class. Refer to the `SampleMetadataBrowser` source file for the abstract class implementation.

A returned instance of `CloudMetadataBrowser` must be capable of:

- Parsing metadata from source artifacts such as WSDL or XML schemas (schemas may be embedded within WSDLs).
- Downloading and parsing metadata from other sources such as a metadata API or a metadata repository within the application.
- Creating the Cloud Application Model from parsed metadata.

### Parsing Source Metadata

Metadata from a data source in the source or target application is parsed into the Cloud Application Model (CAM) for use in the adapter.

Metadata parsing implements the interface `oracle.tip.tools.ide.adapters.cloud.api.metadata.MetadataParser`. This interface parses a resource identified by a `javax.activation.DataSource` into the `CloudApplicationModel`. The `CloudApplicationModel` is an in-memory metadata store. The following objects are stored in the `CloudApplicationModel`. All of these objects are in the package `oracle.tip.tools.ide.adapters.cloud.api.model`.

- **CloudAPINode** — This optional, outbound-only object represents an API that groups operations into logical categories, such as CRUD.
- **CloudOperationNode** — This object represents an operation in a cloud application. Operation nodes contain the following child objects:
  - **OperationResponse** — The Response from an operation.
  - **OperationFault** — A fault or exception that may be raised as a result of invoking this operation.
  - **RequestParameter** — Represents each request parameter. For SOAP-based services using document literal, there is typically only one operation parameter. However, for REST based applications there may be several.
• Header — Represents request header information.

• CloudDataObjectNode — Represents an application business object. For example, in a CRM application, metadata for a Contact is represented.

For parsing metadata, the AbstractMetadataParser class provides two abstract methods:

protected abstract List<MetadataParser> getMetadataParsers()

This method must return a list of parsers that are invoked when metadata parsing is required.

protected abstract void parseMetadata(MetadataParser parser, AdapterPluginContext context)

This method invokes the actual parse method on the MetadataParser, passing in any required parameters from the AdapterPluginContext.

**Parsing Source Metadata from a WSDL File**

For SOAP-based cloud applications, metadata can be parsed from a WSDL file.

The plugin can return an instance of the oracle.tip.tools.ide.adapters.cloud.impl.metadata.wsdl.WSDLMetadataParser in the getMetadataParsers() method.

**Parsing Source Data from an API Call**

Some cloud applications have metadata that is accessible from an API within the application. This metadata is usually extracted as XML content and then parsed into the Cloud Application Model. You must take care of parsing custom or external metadata.

Refer to the SampleMetadataBrowser class under the plugin.metadata package.

<table>
<thead>
<tr>
<th>Package, Class and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.plugin.metadata</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleMetadataBrowser</td>
<td>This class parses metadata information such as operations or business objects.</td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td></td>
</tr>
<tr>
<td>loadMetadata()</td>
<td>Loads metadata information such as business objects and operations.</td>
</tr>
<tr>
<td>getMetadataParsers()</td>
<td>Gets the list of parsers, such as WSDL and SOAP parsers.</td>
</tr>
<tr>
<td>parseMetadata()</td>
<td>Parses the metadata information.</td>
</tr>
</tbody>
</table>
getDataObjects()

Gets data objects. Required for an inbound service definition in which the wizard user can select any data objects as request/reply parameters. A data object is a schema type defined in the XSD file for a business object in the service catalog.

parseWSDLMetadata()

Parses WSDL metadata.

For an outbound adapter, a sample adapter uses CRUD as an API, which groups operations into a logical category.

private static final String CRUD = "CRUD";
// Place to create custom API (CRUD) and assign operations to API
public void processAPINodes() {
    String namespace = SampleCloudAdapterConstants.SAMPLE_SERVICE_NAMESPACE;
    Set<String> operationNames = new HashSet<String>;

    // Gets the list of operations from CAM
    List<CloudOperationNode> opNodes = getModel().getOperations();

    for(int i=0;i<opNodes.size();i++){
        CloudOperationNode opNode = opNodes.get(i);

        // Add operations to API node
        operationNames.add(opNode.getName());
    }

    CloudAPINode api = new CloudAPINodeImpl(CRUD, namespace, getVersion(),
        operationNames);

    // Add API to Cloud Application Model
    getModel().addAPI(api);
}

For an inbound adapter, the metadata browser can hold the complete CAM details
(APIs, Operations, Data objects, and so on). However, the SampleCloudAdapter
Request and Response Configuration Pages display only business objects regardless of
any application operations.

Return the same class (SampleMetadataBrowser) in the

@Override
public CloudMetadataBrowser getMetadataBrowser(CloudConnection connection) {
    AdapterPluginContext context = getPluginContext();
    return new SampleMetadataBrowser(connection,context);
}

Showing APIs
Each CloudAPINode instance has a name and version property that can be displayed.
List<CloudAPINode> apis = browser.getAPIs();

Showing Operations
If there are no APIs returned in getAPIs(), then it’s safe to assume that operations
are not categorized by APIs.
List<CloudOperationNode> operations = browser.getOperations();

If, however, there are APIs returned, then you may want to filter the operations by
API.
//pass in the CloudAPINode instance that was selected by the end user
List<CloudOperationNode> operations = browser.getOperations(selectedAPINode);
//or
Set<String> operationNames = selectedAPINode.getOperationNames();

Showing Data Objects
After selecting an operation, users will typically want to select the data objects that can be submitted to that operation. This list is filtered based on the selected operation. It does not include all objects in the cloud application, only those that are appropriate for use with the selected operation:

```java
List<CloudDataObjectNode> dataObjects = browser.getDataObjectNodes(selectedOperationNode);
```

Each data object has a name, as well as several other properties useful for display.

**Request Headers and Operation Properties**

Some cloud applications declare and expect headers in their request messages. For SOAP applications, headers are transmitted in a SOAPHeader element, whereas REST applications tend to rely on the HTTP headers. In both cases, access to this header information is the same from the standpoint of the UI:

```java
List<Header> headers = selectedOperation.getRequestHeaders();
```

The headers have a hierarchical structure, and the UI represents the structure to the wizard user. Ultimately the user must set values on the leaf nodes (for example, the Field elements) and these must be provided during generation.

### Generating the Runtime Artifacts

The final result of the Design-Time Framework is the generation of runtime artifacts that the Runtime Framework uses.

The `RuntimeMetadataGenerator` interface generates artifacts in specified directories. You can extend `oracle.tip.tools.ide.adapters.cloud.impl.generation.AbstractRuntimeMetadataGenerator`, which contains some default code for processing and invoking the artifact generators.

Extending the abstract class requires implementing only one pure abstract method:

```java
protected abstract void initializeContext(RuntimeGenerationContext context);
```

Subclasses can use this method to add additional objects to the context. In many cases no additional methods need to be overridden when extending `AbstractRuntimeGenerator`. However, some subclasses may need to modify the selected objects or operations. Overriding `setTransformModel` in the generator gives you access to the `TransformationModel`. From the `TransformationModel`, you can access the selected objects and operations.

Refer to the `SampleRuntimeMetadataGenerator` class under the `plugin.generator` package.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.samplecloudadapter.plugin.generator</code></td>
<td></td>
</tr>
</tbody>
</table>
**Class:**

SampleRuntimeMetadataGenerator

This class generates runtime metadata artifacts such as the Cloud Application Configuration file and the integration WSDL.

**Method:**

SampleRuntimeMetadataGenerator()

Initializes class-level variables.

initializeContext()

Similar to setContextObject in a runtime connection factory.

The following sequence diagram shows a simple example of generating runtime artifacts.
Here is a more detailed explanation of the sequence:

1. new TypeMapping(dataObject) — The wizard creates a new oracle.cloud.connector.metadata.TypeMapping for each selected business object.

2. new OperationMapping — The wizard creates a new oracle.cloud.connector.metadata.OperationMapping for each selected operation. These are known as target operations. The OperationMapping instance allows the name of the target operation to be overridden. The overridden name shows in the generated integration WSDL. Typically, only one target operation is selected and thus only one OperationMapping object is needed.

3. setRequestObjectMappings(List<TypeMapping>) — Sets the list of TypeMapping instances (each representing selected business objects) that are the input (or request) for the target operation.

4. setResponseObjectMappings(List<TypeMapping>) — Optionally, the wizard can set mappings for the response. This is unlikely to be necessary, however. More likely, the wizard doesn’t call this method and the plugin implementation handles the response mapping. This is only an illustration of the typical steps for setting up mappings before generation. If no response object mappings are set by the wizard OR the plugin AND the target operation is a synchronous request/response operation, the generator copies the response of the target operation into the generated integration-friendly WSDL.

5. getRuntimeGenerator() — Each plugin provides its implementation of oracle.cloud.connector.generation.RuntimeMetadataGenerator,
which is accessible using a call to getRuntimeGenerator() from the CloudAdapter interface.

6. setConnection(connection) — The RuntimeMetadataGenerator uses various properties in the Connection object to generate the Cloud Application Configuration file. Not all properties are copied. The section on RuntimeMetadataGenerator goes into detail about this process.

7. setTargetNamespace(namespace) — The RuntimeMetadataGenerator uses the target namespace passed to this method as the target namespace of the generated integration-friendly WSDL.

8. setWSDLResourceOutput(output) — The RuntimeMetadataGenerator uses this method to determine where the integration-friendly WSDL is generated.

9. setCloudApplicationConnectionOutput(output) — The RuntimeMetadataGenerator uses this method to determine where the Cloud Application Configuration file is generated.

10. generate() — this method is called by the wizard to generate all artifacts. This method returns an instance of RuntimeMetadata that has the locations of all generated artifacts.

**Runtime Components**

The runtime components connect to the source or target application and manage communication between the adapter and that application.

In the RT framework, each Cloud Application has a corresponding RT component. Each RT component must conform to the Cloud Adapter API, which provides the interfaces that must be implemented. The single entry point for SOA and Service Bus processes is the Cloud Adapter. This entry point is always present in deployments with a Cloud Application Configuration file. The Cloud Application Configuration file contains an attribute that identifies the Cloud Adapter entry point:

```
<connection-factory location="cloud/CloudAdapter"/>
```

The location attribute identifies the entry point. The adapter is only loaded once. Upon invocation, the following runtime flow occurs:

1. The Cloud Connection RT looks up the instance of the Adapter plugin’s RT connection class by checking the value of the managedConnectionFactoryClassName attribute. The instance is only loaded once.

```
<managedConnectionFactoryClassName="oracle.tip.tools.ide.adapters.samplecloudadapter.runtime.SampleCloudAppConnectionFactory"/>
```

2. If the RT instance was not previously loaded, then a new instance is created from the plugin’s implementation of oracle.cloud.adapter.api.CloudConnectionFactory. In addition to this instantiation, various connection level properties are set. Refer to the SampleCloudAppConnectionFactory.java class in the sample code for more details.

3. Once the instance has been retrieved, the next step is to invoke the source or target cloud application operation. The RT Framework gets an instance of oracle.cloud.adapter.api.CloudOperation and calls the invoke method, passing in the request payload.
4. During the invoke call, the plugin’s Operation instance performs all the necessary transformations on the request payload and constructs a payload to send to the cloud application. The RT framework provides special invoker interfaces that can send SOAP, JSON, or XML messages to REST or SOAP endpoints. Refer to the oracle.tip.tools.ide.adapters.samplecloudadapter.runtime.AbstractOperationHandler.java class in the sample code for more details.

If invocation of the cloud application succeeds, the plugin performs any necessary transformations and returns an instance of oracle.cloud.adapter.api.CloudMessage containing the actual response. If the invocation of the cloud application fails, the plugin handles the exception and throws an instance of oracle.cloud.adapter.api.CloudAdapterFault containing the fault details.

For an outbound adapter:

```
Runtime Classes

Cloud Application Configuration File (JCA) ->
SampleCloudAppConnectionFactory
SampleCloudAdapterMessageHandler
AbstractOperationHandler
```

For an inbound adapter:

```
Runtime Classes

Cloud Application Configuration File (JCA) ->
SampleCloudAppConnectionFactory
SampleCloudConnector
SampleMessageReceiver
```

**Endpoint Invoker**

The Runtime Framework connects to and invokes the source or target application. Refer to the SampleCloudAppConnectionFactory runtime package.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.runtime</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleCloudAppConnectionFactory</td>
<td>This class implements the connection factory class during runtime.</td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleCloudAppConnectionFactory()</td>
<td>Initializes class-level variables.</td>
</tr>
<tr>
<td>getConnection()</td>
<td>Gets the connection class (outbound).</td>
</tr>
<tr>
<td>getCloudConnectorClassName()</td>
<td>Gets the connector class name (inbound).</td>
</tr>
</tbody>
</table>
For an outbound adapter, refer to the `SampleCloudAppConnection` runtime package.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.samplecloudadapter.runtime</code></td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td><code>SampleCloudAppConnection</code></td>
<td>This is the actual adapter connection class during runtime. In this class, the MessageHandlers handle request and response payloads.</td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td></td>
</tr>
<tr>
<td><code>getMessageHandlers()</code></td>
<td>Gets message handler classes.</td>
</tr>
</tbody>
</table>

For an inbound adapter, refer to the `SampleCloudConnector` runtime package.
Package:
oracle.tip.tools.ide.adapters.samplecloudadapter.runtime

Class:
SampleCloudConnector

This Connection factory class creates a connector class to invoke an inbound adapter. Request and response handler classes are invoked before a request is submitted to an inbound adapter.

Method:
getCloudConnectorName() Gets the connector name.
getCloudConnectorDescription() Gets the connector description.
getCloudConnectorVersion() Gets the connector version.
getCloudConnectorVendor() Gets the connector vendor details.
eventListenerActivation() Gets the event listener for activation.
eventListenerDeactivation() Gets the event listener for deactivation.

For an inbound adapter, return the CloudMessageReceiver class.

```java
public CloudMessageReceiver
eventListenerActivation(CloudApplicationConnectionFactory
    cloudApplicationConnectionFactory, CloudEndpointFactory cloudEndpointFactory)
throws Exception {
    return new SampleMessageReceiver(cloudApplicationConnectionFactory,
    cloudEndpointFactory);
}
```

// When user deactivates the flow
```java
public void eventListenerDeactivation(CloudMessageReceiver cloudMessageReceiver) {
    cloudMessageReceiver.release();
}
```

Message Transformations for Outbound Adapters

An outbound adapter must often transform messages to and from the target application.

Refer to the SampleCloudAdapterMessageHandler class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package:oracle.tip.tools.ide.adapters.samplecloudadapter.runtime</td>
<td></td>
</tr>
<tr>
<td>Class: SampleCloudAdapterMessageHandler</td>
<td>This class handles incoming requests and adjusts the payload according to the target system.</td>
</tr>
</tbody>
</table>
### Method:

**handleRequestMessage()**
Handles the incoming request and modifies the payload according to the target system.

**handleResponseMessage()**
Handles the response from the target system and delivers it to ICS or JDeveloper.

**handleErrorMessage()**
Handles error messages.

---

Handlers in the sample code handle only the `createAccount()` request payload. (One of the operations in the Sample Account service is `createAccount()`.)

```java
private static String CREATE_OPERATION = "createAccount";
private Map<String, OperationHandler> operationHandlers = new HashMap<String, OperationHandler>();

public SampleCloudAdapterMessageHandler() {
    operationHandlers.put(CREATE_OPERATION, new CreateOperationHandler());
}
```

You can implement the `handleRequestMessage` method to adjust the payload according to the target system schema.

```java
// Gets the appropriate message handler based on operation selection in UI
OperationHandler handler = operationHandlers.get(targetOperationName.getLocalPart());

// Creating SOAPHeader according to target endpoint
MessageHeader header = createWSSHeader(jcaEndpointInteractionProperties, authManager);
message.addMessageHeader(header);
```
You can implement `handleResponseMessage` in a format understandable to ICS or JDeveloper.

```java
// Gets the appropriate response message handler based on selection
OperationHandler handler = operationHandlers.get(context.getTargetOperationName());
```

Here are details of the `AbstractOperationHandler` abstract class:

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package:</td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.samplecloudadapter.runtime</code></td>
<td></td>
</tr>
<tr>
<td>Class:</td>
<td></td>
</tr>
<tr>
<td><code>AbstractOperationHandler</code></td>
<td>This abstract class normalizes the payload.</td>
</tr>
<tr>
<td>Method:</td>
<td></td>
</tr>
<tr>
<td><code>handleOperationRequest()</code></td>
<td>Handles the incoming request to normalize the root element. ICS or JDeveloper generates the operation that holds the data object of the target system service.</td>
</tr>
<tr>
<td><code>handleOperationResponse()</code></td>
<td>Handles the response.</td>
</tr>
<tr>
<td><code>handleOperationError()</code></td>
<td>Handles error messages.</td>
</tr>
</tbody>
</table>

Here is a basic implementation of the `handleOperationRequest` method.

```java
// Normalize root element is to change integration WSDL specification to concrete WSDL spec.
Element sourceRootElement = normalizeRootElement(context, requestDocument, version);
```

The `CreateOperationHandler` class extends `AbstractOperationHandler` to handle messages in case of any further changes.

```java
public class CreateOperationHandler extends AbstractOperationHandler {
}
```

### Message Transformations for Inbound Adapters

An inbound adapter must often transform messages to and from the source application.

Refer to the `SampleMessageReceiver` runtime package.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package:</td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.samplecloudadapter.runtime</code></td>
<td></td>
</tr>
<tr>
<td>Class:</td>
<td></td>
</tr>
<tr>
<td><code>SampleMessageReceiver</code></td>
<td>This class handles the request and response messages before submitting them to the inbound adapter.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>onMessage()</td>
<td>The SOAP Listener invokes this method to transform the request</td>
</tr>
<tr>
<td></td>
<td>from an external WSDL to a local WSDL.</td>
</tr>
<tr>
<td>run()</td>
<td>Runs the event listener.</td>
</tr>
<tr>
<td>activation()</td>
<td>Handles integration flow activation.</td>
</tr>
<tr>
<td>release()</td>
<td>Handles integration flow deactivation.</td>
</tr>
<tr>
<td>handleResponseMessage()</td>
<td>Handles the response message.</td>
</tr>
<tr>
<td>handleRequestMessage()</td>
<td>Handles the request message.</td>
</tr>
</tbody>
</table>
Runtime Components

```
<<implementation class>>
SampleMessageReceiver

- m_cloudEndpointFactory: CloudEndpointFactory
- m_cloudEndpoint: CloudEndpoint
- m_logger: CloudAdapterLoggingService
- m_isReleased: boolean
- m_cloudInvocationContext: CloudInvocationContext
- m_requestWsdlTargetNS: string

+ SampleMessageReceiver(in connFactory:
  CloudApplicationConnectionFactory, in cloudEndpointFactory:
  CloudEndpointFactory)
+ onMessage(in cloudRequestMessage: CloudMessage)
+ activation()
+ release()
+ handleRequestMessage(in cloudRequestMessage:
  CloudInvocationException)
+ handleResponseMessage(in cloudResponseMessage: CloudMessage)
+ getEndpointWsdl()
+ getEndpointWsdlDef()
```

Cloud Adapter Plugin SDK

- BaseSOAPMessageReceiver
- CloudApplicationConnectionFactory
- CloudAdapterLoggingService
- CloudInvocationContext
- CloudConnectorException
- CloudEndPointFactory
- CloudInvocationException
- CloudMessage
- CloudEndpoint

extends

uses
Implement the `onMessage()` method. Message transformation is required only when the inbound message is polymorphic.

```java
// Submit request to inbound adapter service
CloudMessage cloudResponseMessage = m_cloudEndpoint.raiseEvent(cloudRequestMessage);
```
Developing the Sample REST Cloud Adapter

This chapter describes the process of developing a cloud adapter named REST Cloud Adapter based on a sample integration service. See the SampleRESTAdapter project for more detailed source code.

Topics

• About REST Cloud Adapters
• Creating a Metadata Catalog for a REST Cloud Adapter
• Setting Up the REST Cloud Adapter Development Environment
• Creating a cloud-adapter.xml File for a REST Cloud Adapter
• Creating Outbound REST Cloud Adapter Design-Time Components
• Creating Outbound REST Cloud Adapter Runtime Components
• Creating Inbound REST Cloud Adapter Components

About REST Cloud Adapters

REST Cloud adapters allow easy calling and exposing of RESTful services based on a WADL or manual configuration. Use of JSON (JavaScript Object Notation) files is supported.

How REST Adapters Are Different

REST Cloud adapters differ from web-service-based adapters in these ways:

• Some of the library JAR files on which REST adapters depend are different.
• The cloud-adapter.xml file has some REST-specific values.
• No WSDL files are generated.
• The wizard has only Welcome, Operations, and Summary pages. A Connection page is unnecessary because each REST operation connects separately.
• The metadata parser requires a pre-created metadata catalog.
• Message transformations are different, involving conversion to and from JSON.

Prerequisites

Before you can develop a REST adapter, you must:
• Have access to the REST API for which you are creating an adapter
• Create a single metadata catalog for all the REST resources
• Know the authentication type and media type of each REST resource

Sample REST Application and REST Adapter
A sample REST application to which the sample REST adapter can connect is provided. A web application called SampleRestApplication.war has REST resources such as account and contact and documentation artifacts. Each individual resource returns link data to identify the resource and all supported services.

Configured as a source or target in ICS, the sample REST adapter can connect to any external REST-exposed SaaS application. You can use the sample REST application to test it.

Creating a Metadata Catalog for a REST Cloud Adapter
Creating a metadata catalog for the REST application to which your REST adapter will connect is one of the prerequisites for creating a REST adapter.

A sample metadata catalog file, named samplemetadatacatalog.json, is based on the REST resources in the sample REST API. It contains a list of resource details and corresponding metadata information for each resource, including JSON schemas. The file looks like this:

```json
{
  "items": [
    {
      "name": "account",
      "links": [
        {
          "href": "file:///account.json",
          "rel": "describes",
          "mediaType": "application/schema+json"
        }
      ]
    },
    {
      "name": "contact",
      "links": [
        {
          "href": "file:///contact.json",
          "rel": "describes",
          "mediaType": "application/schema+json"
        }
      ]
    }
  ]
}
```

Setting Up the REST Cloud Adapter Development Environment
The Sample REST Cloud Adapter project (SampleRESTAdapterProj) has dependencies upon the following JAR files, which are required for successful compilation. You can set up dependency libraries in the project properties to use these JAR files from the ORACLE_HOME location. Or you can extract these JAR files from ORACLE_HOME subdirectories. An example ORACLE_HOME location is C:\Oracle\Middleware\ORACLE_HOME.
<table>
<thead>
<tr>
<th>No</th>
<th>Name of the Library</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cloud-designtime-api.jar</td>
<td>ORACLE_HOME\soa\soa\modules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\oracle.cloud.adapter_12.1.3</td>
</tr>
<tr>
<td>2</td>
<td>cloud-designtime-impl.jar</td>
<td>ORACLE_HOME\soa\soa\modules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\oracle.cloud.adapter_12.1.3</td>
</tr>
<tr>
<td>3</td>
<td>cloud-runtime-api.jar</td>
<td>ORACLE_HOME\soa\soa\modules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\oracle.cloud.adapter_12.1.3</td>
</tr>
<tr>
<td>4</td>
<td>cloud-runtime-impl.jar</td>
<td>ORACLE_HOME\soa\soa\modules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\oracle.cloud.adapter_12.1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ORACLE_HOME\soa\soa\modules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\oracle.cloud.adapter_12.1.3</td>
</tr>
<tr>
<td>5</td>
<td>oracle.tools.cloud.adapter.sdk.jar</td>
<td>ORACLE_HOME\osb\lib\uitools</td>
</tr>
<tr>
<td>6</td>
<td>oracle.tools.uiobjects.sdk.jar</td>
<td>ORACLE_HOME\osb\lib\uitools</td>
</tr>
<tr>
<td>7</td>
<td>oracle.tools.cloud.adapter.ide.jar</td>
<td>ORACLE_HOME\osb\plugins\jdeveloper</td>
</tr>
<tr>
<td></td>
<td></td>
<td>extensions</td>
</tr>
<tr>
<td>8</td>
<td>com.oracle.webservices.orawsdl-api_12.1.3.jar</td>
<td>ORACLE_HOME\oracle_common\modules</td>
</tr>
<tr>
<td>9</td>
<td>bpm-infra.jar</td>
<td>ORACLE_HOME\soa\soa\modules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\oracle.soa.fabric_11.1.1</td>
</tr>
<tr>
<td>10</td>
<td>jersey-core-1.18.jar</td>
<td>ORACLE_HOME\oracle_common\modules</td>
</tr>
</tbody>
</table>

**Creating a cloud-adapter.xml File for a REST Cloud Adapter**

To enable adapter registration, a REST-based adapter plugin MUST contain an XML file named cloud-adapter.xml in the META-INF folder of the plugin JAR file.

To create a project for a REST Cloud adapter, follow the steps in Creating a cloud-adapter.xml File for a Cloud Adapter. However, note that the cloud-adapter.xml file for a REST adapter is slightly different. The main differences are in the adapterFactory, UIProviderClass, and resourceBundle values. Here is the cloud-adapter.xml file for the Sample REST Adapter:

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<tns:AdapterPluginConfig xmlns:tns=http://xmlns.oracle.com/adapters/cloud>
  <adapterPluginID>RESTADAPTERPLUGINID</adapterPluginID>
  <displayName>${nls.plugin.displayName}</displayName>
  <description>${nls.plugin.description}</description>
  <adapterFactory>oracle.tip.tools.ide.adapters.cloud.rest.sample.plugin.SampleRestAdapterFactory</adapterFactory>
  <messageExchangePatterns>
    <messageExchangePattern>REQUEST_REQUEST</messageExchangePattern>
  </messageExchangePatterns>
</tns:AdapterPluginConfig>
```
Creating Outbound REST Cloud Adapter Design-Time Components

Outbound REST-based adapters have design-time components similar to those of other custom outbound adapters, including a User Interface Module and a Plugin Module.

The following diagram shows the design-time components in an outbound REST-based adapter.
The design-time plugin developer must provide concrete implementations of the following interfaces:

- **CloudApplicationAdapterFactory** — The interface `com.oracle.cloud.connector.CloudApplicationAdapterFactory` creates instances of the `CloudApplicationAdapter`. Adapter developers must provide an instance of this interface. However, although this is a required interface, some wizards don’t use it. For example, the JDeveloper Adapter Wizard Framework does not use this interface to create instances of the adapter.

- **CloudApplicationAdapter** — An implementation of this interface provides an entry point to the design-time adapter plugin. From this interface, a wizard gets handles to other interfaces such as `CloudMetadataBrowser`, `RuntimeMetadataGenerator`, and so on.

The design-time framework can leverage existing default interface implementations, which provide baseline functionality without requiring explicit realization of all required interfaces.

- **DefaultCloudConnection** — This is the default implementation of the `CloudConnection` interface. It stores user-provided connection information, such as username, password, and WSDL location.
• **DefaultMetadataBrowser** — You can return an instance of this class in response to the `getMetadataBrowser()` method in the mandatory `CloudApplicationAdapter` interface. The default implementation can automatically provide metadata information if an instance of the `CloudConnection` interface exists.

• **DefaultRuntimeGenerator** — This class provides a default implementation of the `RuntimeGenerator` interface. It generates runtime artifacts if a `TransformationModel` and an instance of `CloudConnection` exist. The `getRuntimeGenerator()` method of the `CloudApplicationAdapter` interface can return an instance of this class.

All default implementations extend abstract classes prefixed with **Abstract**, such as `AbstractCloudConnection`. Design-time plugins can extend abstract classes directly or extend the default implementations. The former is recommended, because the default implementations are internal classes and subject to change, while the abstract classes are part of the API and are therefore more stable.

Refer to the `SampleRestAdapterFactory` class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td><strong>oracle.tip.tools.ide.adapters.cloud.rest.sample.plugin</strong></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td><strong>SampleRestAdapterFactory</strong></td>
</tr>
<tr>
<td><strong>createAdapter()</strong></td>
<td>Returns an adapter instance.</td>
</tr>
</tbody>
</table>

Refer to the `SampleCloudAdapter` class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td><strong>oracle.tip.tools.ide.adapters.cloud.rest.sample.plugin</strong></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td><strong>SampleCloudAdapter</strong></td>
</tr>
<tr>
<td><strong>getConnection()</strong></td>
<td>Returns the adapter’s connection object.</td>
</tr>
<tr>
<td><strong>getMetadataBrowser()</strong></td>
<td>Returns the adapter’s metadata browser.</td>
</tr>
<tr>
<td><strong>getRuntimeMetadataGenerator()</strong></td>
<td>Returns the adapter’s <code>MetadataGenerator</code> class.</td>
</tr>
<tr>
<td><strong>getName()</strong></td>
<td>Returns the name of the adapter.</td>
</tr>
</tbody>
</table>

Refer to the `SampleCloudAdapterConnection` class.
<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.cloud.rest.sample.plugin</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleCloudAdapterConnection</td>
<td>This class lets the wizard user test the adapter’s connectivity.</td>
</tr>
<tr>
<td>ping()</td>
<td>Tests connectivity to the target application.</td>
</tr>
<tr>
<td>getAuthenticationScheme()</td>
<td>Stores authentication credentials.</td>
</tr>
</tbody>
</table>

Refer to the SampleMetadataBrowser class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.cloud.rest.sample.plugin.metadata</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleMetadataBrowser</td>
<td>This class parses the metadata of the target application so it can be displayed and selected in the wizard. In this case, it creates a metadata catalog for REST services. The metadata browser uses the Cloud Application Model (CAM).</td>
</tr>
<tr>
<td>getMetadataParsers()</td>
<td>Returns a list of metadata parsers. In this case, the MetadataCatalogParser is returned.</td>
</tr>
<tr>
<td>parseMetadata()</td>
<td>Parses the metadata. In this case, the JSON metadata catalog is parsed.</td>
</tr>
</tbody>
</table>

Refer to the SampleRuntimeMetadataGenerator class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.samplecloudadapter.plugin.generator</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleRuntimeMetadataGenerator</td>
<td>This class generates runtime metadata artifacts such as the Cloud application configuration file, WSDL files, and so on.</td>
</tr>
<tr>
<td>SampleRuntimeMetadataGenerator()</td>
<td>Initializes class-level variables.</td>
</tr>
</tbody>
</table>
**initializeContext()**

Sets context objects such as the runtime connection factory.

Refer to the **SampleRestAdapterProvider** class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.cloud.rest.sample.wizard</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleRestAdapterProvider</td>
<td>This class creates the adapter-specific UI binding. Every adapter should extend the CloudAdapterUIProvider abstract class.</td>
</tr>
</tbody>
</table>

**getCloudAdapterUIBinding()**

Returns a new instance of the adapter-specific UI binding.

**getLocalizedAdapterType()**

Returns the adapter type, which is used in the wizard header.

Refer to the **SampleCloudAdapterUIBinding** class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.cloud.rest.sample.wizard</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class:</th>
<th>Description</th>
</tr>
</thead>
</table>
| SampleCloudAdapterUIBinding | This class implements ICloudAdapterUIBinding and:  
  - Creates the DefaultAdapterPlugin Context. This adapter-specific context creates artifacts. This context object is also passed to all pages, which can use this for passing values from one page to other.  
  - Gets the list of pages, and UI fields for each page. Performs business logic validation, delegates page actions to respective pages, and generates artifacts. Handles interaction between the wizard and the actual adapter pages. |
getEditPages() Returns the list of pages to be displayed in the wizard. The key in the map is pagId, and the value is an instance of ICloudAdapterPage. The sequence of pages in the map determines the sequence in the wizard.

generateMetadataArtifacts() Returns CloudAdapter runtime artifacts based on user selections. Runtime artifacts include the Cloud adapter configuration file and the integration WSDL.

Refer to the SampleRestAdapterWelcomePage class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong> oracle.tip.tools.ide.adapters.cloud.rest.sample.wizard</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong> SampleRestAdapterWelcomePage</td>
<td>This class implements ICloudAdapterPage and creates the Welcome page in the wizard.</td>
</tr>
<tr>
<td>getWelcomeText()</td>
<td>Returns welcome text, which briefly describes the adapter to the wizard user.</td>
</tr>
</tbody>
</table>

This class can inherit the following API to modify the default WelcomePage properties. Here is a code snippet that shows how to extend CloudAdapterWelcomePage and change the Welcome Page text details.

```java
import oracle.tip.tools.adapters.cloud.impl.CloudAdapterWelcomePage;

public class SampleCloudAdapterWelcomePage extends CloudAdapterWelcomePage {
    public String getWelcomeText() {
        return SampleCloudAdapterCloudText.getString(SampleCloudAdapterCloudText.WELCOME_PAGE_WELCOME_TEXT);
    }
}
```

Refer to the SampleCloudAdapterOperationsPage class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong> oracle.tip.tools.ide.adapters.cloud.rest.sample.wizard</td>
<td></td>
</tr>
</tbody>
</table>
This class implements ICloudAdapterPage and creates the Operations page in the wizard. The Operations page displays application-specific business operations and corresponding data objects.

- **getPageId()**  
  Gets the page ID and initializes class-level variables.

- **getPageName()**  
  Gets the page name.

- **getWelcomeText()**  
  Gets the page’s welcome text.

- **getPageTitle()**  
  Gets the page title.

- **getPageEditFields()**  
  Returns the page fields.

- **getUpdatedEditPages()**  
  Updates the page controls based on user selections.

- **updateBackEndModel()**  
  Updates the back-end data model.

The following code snippet displays operations and business objects.

```java
try { browser = CloudAdapterUtils.getMetadataBrowser(adapterPluginContext); }  
catch(Exception e) { e.printStackTrace(); }

// display list of resources  
String defaultResourceName = configureResources(browser, fields);

// display list of resource operations  
configureOperations(defaultResourceName, fields);
```

Refer to the SampleRestAdapterSummaryPage class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.cloud.rest.sample.wizard</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleRestAdapterSummaryPage</td>
<td>This class implements ICloudAdapterPage and creates the Summary page in the wizard. The Summary page displays the configuration information that the user selected on previous pages.</td>
</tr>
<tr>
<td><strong>getWelcomeText()</strong></td>
<td>Gets the page’s welcome text.</td>
</tr>
<tr>
<td><strong>getPageEditFields()</strong></td>
<td>Returns the page fields.</td>
</tr>
</tbody>
</table>
This page inherits and extends the UI framework’s default Summary page properties. You can extract and display information from the adapter context as part of the configuration summary.

```java
public class SampleRestAdapterSummaryPage extends CloudAdapterSummaryPage
```

## Creating Outbound REST Cloud Adapter Runtime Components

Outbound REST-based adapters have runtime components similar to those of other custom outbound adapters, including an Endpoint Invoker and message transformations.

The following diagram shows the runtime components in the sample outbound REST-based adapter.

![Diagram showing runtime components]

For general information about the interfaces and classes in the runtime environment, see [Runtime Components](#).

Refer to the `SampleRestAdapterConnectionFactory` class.

### Package, Class, and Methods

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package</strong>: oracle.tip.tools.ide.adapters.cloud.rest.sample.runtime</td>
<td></td>
</tr>
<tr>
<td><strong>Class</strong>: SampleRestAdapterConnectionFactory</td>
<td>This class lets you implement a connection factory class during runtime.</td>
</tr>
<tr>
<td>SampleRestAdapterConnectionFactory()</td>
<td>Initializes class-level variables.</td>
</tr>
</tbody>
</table>
getConnection() Returns the connection class.

Refer to the SampleRestAdapterRuntimeConnection class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.cloud.rest.sample.runtime</code></td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleRestAdapterRuntimeConnection</td>
<td>This class is the actual adapter connection class during runtime. In this class, MessageHandler objects are created to handle request and response payloads.</td>
</tr>
<tr>
<td>getMessageHandlers()</td>
<td>Returns message handler classes.</td>
</tr>
</tbody>
</table>

Refer to the SampleRestAdapterMessageHandler class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td><code>oracle.tip.tools.ide.adapters.cloud.rest.sample.runtime</code></td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleRestAdapterMessageHandler</td>
<td>This class handles incoming requests and adjusts the payload according to the target system.</td>
</tr>
<tr>
<td>handleRequestMessage()</td>
<td>Handles incoming requests and adjusts the payload according to the target system.</td>
</tr>
<tr>
<td>handleResponseMessage()</td>
<td>Handles the response from target system and delivers it to ICS.</td>
</tr>
<tr>
<td>handleErrorMessage()</td>
<td>Handles error messages.</td>
</tr>
</tbody>
</table>

A message in an ICS flow is always in XML format. A message from an inbound REST adapter is converted from JSON or URL-encoded format to XML using NXSD. Similarly, an XML message to an outbound REST adapter is converted using NXSD to JSON or URL-encoding depending on the configured media type. The following diagram shows these message transformations:

You can transform an incoming request message from XML to the target application content type:

```java
addHeaderToCloudMessage(cloudMessage, cloudInvocationContext);
translateRequestMessage(cloudInvocationContext, (RESTCloudMessage) cloudMessage);
```
Creating Inbound REST Cloud Adapter Components

Inbound REST-based adapters have components similar to those of other custom inbound adapters.

An inbound adapter receives and validates requests from external systems, also called client applications. No change to the client application code is needed, because the adapter handles calls that are already native to the client application.

Make sure the `MessageExchangePattern` property in the `cloud-adapter.xml` file is set to `NOTIFICATION`. This property enables adapters to be inbound.

```xml
<messageExchangePattern>NOTIFICATION</messageExchangePattern>
```

Refer to the `SampleRestAdapterUIBinding` class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td>oracle.tip.tools.ide.adapters.cloud.rest.sample.wizard</td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td>SampleRestAdapterUIBinding</td>
</tr>
<tr>
<td></td>
<td>This class implements <code>ICloudAdapterUIBinding</code> and:</td>
</tr>
<tr>
<td></td>
<td>• Creates the `DefaultAdapterPlugin Context. This adapter-specific context creates artifacts. This context object is also passed to all pages, which can use this for passing values from one page to other.</td>
</tr>
<tr>
<td></td>
<td>• Gets the list of pages, and UI fields for each page. Performs business logic validation, delegates page actions to respective pages, and generates artifacts. Handles interaction between the wizard and the actual adapter pages.</td>
</tr>
</tbody>
</table>

```java
SampleRestAdapterUIBinding() 
Initializes the adapter and its class-level variables.
```

The following code snippet sets the inbound flag:

```java
// CloudAdapterFilter SDK class is initialized based on interaction (Inbound / Outbound)
cloudAdapterFilter.isInbound()
```

The following code snippet sets the inbound status in the adapter context:
// Set inbound status in adapter context so status can be leveraged throughout
design time
context.setContextObject(CloudAdapterConstants.UI_IS_INBOUND,
Boolean.valueOf(cloudAdapterFilter.isInbound()));

Refer to the SampleRestAdapterUIUtil class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.cloud.rest.sample.wizard</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleRestAdapterUIUtil</td>
<td>This class provides UI utility functions for design time.</td>
</tr>
<tr>
<td>setOperationMappingsForGenerator()</td>
<td>Sets operation mapping for both inbound and outbound interaction. Selected Operation and Business Object (Request and Response) fields are reflected in the Cloud Application Configuration (.jca) file.</td>
</tr>
</tbody>
</table>

The following code snippet assigns properties and corresponding values based on outbound and inbound interactions:

```java
if((Boolean)context.getContextObject(CloudAdapterConstants.UI_IS_INBOUND))
    setOperationMappingForInbound(modelBuilder, context);
else
    setOperationMappingForMetadataGenerator(modelBuilder, context);
```

Refer to the SampleRestAdapterConnectionFactory class.

<table>
<thead>
<tr>
<th>Package, Class, and Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Package:</strong></td>
<td></td>
</tr>
<tr>
<td>oracle.tip.tools.ide.adapters.cloud.rest.sample.runtime</td>
<td></td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td></td>
</tr>
<tr>
<td>SampleRestAdapterConnectionFactory</td>
<td>This class lets adapter developers implement a connection factory class during runtime.</td>
</tr>
<tr>
<td>getCloudConnectorClassName()</td>
<td>Returns the CloudConnector class, which is required for inbound interactions.</td>
</tr>
</tbody>
</table>

The following code snippet creates the CloudConnector class in the ConnectionFactory class:

```java
@Override
public String getCloudConnectorClassName() {
    return SampleRESTAdapterCloudConnector.class.getName();
}
```

Refer to the SampleRESTAdapterCloudConnector class.
### Package, Class, and Methods

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
</table>
| oracle.tip.tools.ide.adapters.cloud.rest.sample.runtime | SampleRESTAdapterCloudConnector  
This class creates a connector class to invoke an inbound adapter. Request and response handler classes are invoked before a request is submitted to an inbound adapter. |
| | eventListenerActivation()  
This event listener method is invoked during integration flow activation in ICS. In the sample REST implementation, this method creates a Handler object to handle incoming requests. |

The following code snippet creates the Message Handler class in the CloudConnector implementation class:

```java
public CloudMessageReceiver eventListenerActivation(CloudApplicationConnectionFactory connectionFactory, CloudEndpointFactory endpointFactory)
    throws Exception
{
    return new SampleRESTMessageReceiver(connectionFactory, endpointFactory);
}
```

Refer to the `SampleRESTMessageReceiver` class.

### Package, Class, and Methods

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
</table>
| oracle.tip.tools.ide.adapters.cloud.rest.sample.runtime | SampleRESTMessageReceiver  
This class handles the request and response messages before submitting them to the target system. Handlers are required to translate request messages from JSON to XML format and response messages from XML to JSON format. |
| | onMessage()  
Message translation takes place in this method. |
| | translateRequestMessage()  
Uses the SDK Util class to translate incoming messages from JSON to XML format. |
| | translateResponseMessage()  
Uses the SDK Util class to translate response messages from XML to JSON format. |
Use the following URL format to display metadata information for an inbound REST adapter in ICS:

https://host:ssl-port/integration/flowapi/adapter-plugin-id/integration-name/v01/metadata

Use the following URL format to access a REST resource exposed by an inbound REST adapter in ICS:


To build, package, and test the REST Cloud Adapter, see Building and Packaging the Sample Cloud Adapter.
Building and Packaging the Sample Cloud Adapter

This chapter describes the process of building and packaging the Sample Cloud Adapter and registering it with JDeveloper or Integration Cloud Service.

Topics

- Building the Sample Cloud Adapter
- Packaging and Generating the JAR File
- Registering the Adapter
- Testing an Outbound Adapter
- Testing an Inbound Adapter

Building the Sample Cloud Adapter

Build the Sample Cloud Adapter application by following these steps.

1. Right click on the Sample Cloud Adapter project and select Rebuild Sample Cloud Adapter.

   This option builds the project and displays the project status.
2. Ensure that there are no errors in the project.

Packaging and Generating the JAR File

Package the JAR file by following these steps.

You create a JDeveloper deployment profile to generate the Sample Cloud Adapter JAR file. The following diagram shows the content of the JAR file.
1. Right click on the Sample Cloud Adapter project and select **Deploy > New Deployment Profile**.

   This option builds the project and displays its status.
2. Select the JAR file from the Profile Type list and type a Deployment Profile Name.

3. Include the custom MANIFEST.MF file, because this project has design-time and runtime SDK class dependencies. Add the manifest file from the project, located at `src\META-INF\MANIFEST.MF`. 
4. Generate the JAR file from the project. Right click the project and select Deploy > deployment profile name, in this case samplecloudadapter.

5. Click Finish to generate the JAR file under the project (SampleCloudAdapter) deployment folder.
Registering the Adapter

This section guides you through the registration process for both design time and runtime.

Registering the Sample Cloud Adapter with JDeveloper

1. For Design Time, copy the newly generated adapter bundle (JAR) to this location: %ORACLE_HOME%/jdeveloper/dropins.

2. For Runtime, add the adapter JAR file to the CLASSPATH by editing the setDomainEnv script file. This file is in the bin subdirectory of the domain directory, WebLogicInstallDirectory\user_projects\domains\DomainName\bin. Here is an example entry in the file if JDeveloper runs on Windows:

   set POST_CLASSPATH=%ORACLE_HOME%/soa/soa/modules/oracle.cloud.adapter_12.1.3/samplecloudadapter.jar;%POST_CLASSPATH%

   Here is an example entry in the file if JDeveloper runs on Linux:

   POST_CLASSPATH="$ORACLE_HOME/soa/soa/modules/oracle.cloud.adapter_12.1.3/samplecloudadapter.jar:$CLASSPATHSEP:$POST_CLASSPATH"
   export POST_CLASSPATH

3. Restart the WebLogic SOA server.

Registering the Sample Cloud Adapter with ICS

1. For Design Time, copy the newly generated adapter bundle (JAR) to this location: ORACLE_HOME/soa/soa/modules/oracle.cloud.adapter_12.1.3.

2. For Runtime, add the adapter JAR file to the CLASSPATH by editing the setDomainEnv script file. This file is in the bin subdirectory of the domain
directory, WebLogicInstallDirectory/user_projects/domains/DomainName/bin. Here is an example entry in the file:

POST_CLASSPATH= "ORACLE_HOME/soa/soa/modules/oracle.cloud.adapter_12.1.3/samplecloudadapter.jar$(CLASSPATHSEP){POST_CLASSPATH}"
export POST_CLASSPATH

3. Restart the WebLogic SOA server.

Testing an Outbound Adapter

To test an outbound adapter in JDeveloper, add it to an application as an external reference, then deploy and run the application.

Topics

- Creating an Application for Testing an Outbound Adapter
- Adding an External Reference and Using the Adapter Wizard
- Creating a BPEL Client to Invoke the Adapter
- Deploying the Application that Tests the Outbound Adapter
- Testing the Outbound Adapter Using the EM Console

Creating an Application for Testing an Outbound Adapter

To test an adapter, you must add it to an application. So the first task is to create a new application.

1. Open JDeveloper.
2. Select File > New > From Gallery.
3. Select **SOA Application** from the gallery.

4. Type an Application Name.

5. Type a Project Name.

6. Click **Finish.**
An empty `composite.xml` file is created for the test project.
See Adding an External Reference and Using the Adapter Wizard.

**Adding an External Reference and Using the Adapter Wizard**

In JDeveloper, you add an outbound adapter to an application as an external reference.

1. Drag and drop the SampleCloudAdapter from the Cloud category of the component palette to the External References section.

2. The wizard pops up so you can configure the SampleCloudAdapter. The wizard starts with the Welcome page, which displays the SampleCloudAdapter reference name.
3. Click **Next**. The Connection page appears and asks you to provide the target WSDL reference.

You can also enter security credentials using an Authentication Key.

4. Click **Test** to test the connectivity to the service.
A success or failure message is displayed in the Text area.

5. Click **Next**. The Operations Page appears and asks you to select the Operation and its corresponding Business Object.

6. Move the desired business object from the Available list to the Selected list.

7. Click Next. The Summary page appears and displays the items selected during configuration.
8. Click **Finish** to generate the runtime artifacts and references.

**Note:** If clicking **Finish** results in a NullPointerException in the UI thread, it means that you didn’t implement the `updateBackEndModel()` method for one or more of your wizard pages.

These files are generated:

- `SampleReference.jca` (Cloud Application Configuration File)
- `SampleReference.wsdl` (Integration WSDL)

See [Creating a BPEL Client to Invoke the Adapter](#).
Creating a BPEL Client to Invoke the Adapter

Within the adapter testing application, a BPEL client invokes the adapter.

1. Create a BPEL client to invoke the SampleReference.

2. Configure the BPEL client.

3. Wire the BpelClientForSampleCloudAdapter to the SampleReference.

   Note that the blepclientforsamplecloudadapter service was created and wired for you when you created the BPEL client.
4. Double click the BpelClient component to open the BPEL process flow.

5. After the receiveInput activity, drag and drop the Invoke activity from the Webservice component to call the Service reference.

6. Wire Invoke to SampleReference to make the web service connection.

7. Configure the Invoke activity and click OK to establish the connection.
8. To map and transform the request values from source to destination, add an Assign activity above the Invoke activity.

9. Double click the Assign activity to transform the values.
10. Map the input value from source (client:input string) to destination (OrganizationName).

11. Click OK.

12. To map response values, drag and drop a second Assign activity after the Invoke activity.

13. Map the target system output variable (PartyId) to the adapter output variable (client:result).
14. Save all modified files in JDeveloper.

JDeveloper builds the BPEL Client project.

15. Make sure all validation checks pass.

See Deploying the Application that Tests the Outbound Adapter.

Deploying the Application that Tests the Outbound Adapter

You must deploy the application that uses the adapter before you can test the adapter.

1. Configure the Weblogic SOA server and start the server instance.

   The server is named IntegratedWeblogicServer in this example.
2. Right click Project > Deployment > New Deployment Profile…

3. For the Profile Type, select SOA-SAR File, and type a Deployment Profile Name.

![Create Deployment Profile dialog box](image)

4. Accept the default values for the other options and click OK.

5. After the IntegratedWeblogicServer starts, Right click Project > Deployment > Profile-Name.

6. Select Deploy to Application Server and click Next.

7. Use default values or change the Revision ID.
8. Make sure to select this option: **Overwrite any existing composites with the same revision ID.** This redeploys to the same version of a project.

9. Select **IntegratedWeblogicServer** and click **Next**.

10. Make sure the server is running and that the project is deployed in the correct partition.

11. Click **Next** to see a summary of your selections.

12. Click **Finish**.

See Testing the Outbound Adapter Using the EM Console.

**Testing the Outbound Adapter Using the EM Console**

After the application that uses the adapter is deployed, you can run it in the Enterprise Manager (EM) Console.

1. Open the EM Console after successful deployment.
   
   The URL is `http://hostname:port/em`.

2. Enter your credentials and log into the EM Console.

3. Expand the `soa-infra` node and expand the default domain.

4. Select the adapter testing application and click the **Test** button.

5. Expand the Request Payload and enter the `OrganizationName` value.
6. Click **Test Web Service** (in the top right corner).

The PartID is returned from the Cloud Service. You mapped the PartID unique reference number from the response payload in **Creating a BPEL Client to Invoke the Adapter**.

**Testing an Inbound Adapter**

To test an inbound adapter in JDeveloper, add it to an application as an exposed service, then deploy and run the application.

**Topics**

- Creating an Application for Testing an Inbound Adapter
Creating an Application for Testing an Inbound Adapter

To test an adapter, you must add it to an application. So the first task is to create a new application.

1. Open JDeveloper.
2. Select File > New > Gallery.
3. Select SOA Application from the gallery.
4. Type an Application Name.
5. Type a Project Name.

6. Click Finish.

An empty composite.xml file is created for the Test project.

See Adding an Exposed Service and Using the Adapter Wizard.
Adding an Exposed Service and Using the Adapter Wizard

In JDeveloper, you add an inbound adapter to an application as an exposed service.

1. Drag and drop the SampleCloudAdapter from the component palette (Cloud Category) to the Exposed Services section.

2. The wizard pops up and asks you to configure inbound-related information. The wizard starts with the Welcome page, which asks you to type a SampleCloudAdapter reference name.

3. Click Next. The Connection page appears and asks you to provide the source WSDL reference.

   You can also enter security credentials using an Authentication Key.
4. To test connectivity to the service, click **Test**.

A success or failure message is displayed in the Text area.

5. Click **Next**. Choose the business object you would like to receive from the Oracle Sample Cloud Service as the request for this integration flow.
6. Choose the business object you would like to receive from Oracle Sample Cloud Service as the response from this integration flow.

7. Choose **Immediate** for the synchronous integration pattern.

The Immediate response mode specifies a conversational, synchronous response, while the Delayed mode specifies an asynchronous response.
8. Click **Next**. The Summary page appears and displays the items selected during configuration.

![Screen shot](image)

9. Click **Finish** to generate the runtime artifacts and references.

**Note:** If clicking **Finish** results in a `NullPointerException` in the UI thread, it means that you didn’t implement the `updateBackEndModel()` method for one or more of your wizard pages.
These files are generated:

- SampleService_REQUEST.jca (Cloud Application Configuration File)
- SampleService_REQUEST.WSDL (Integration WSDL)

One of the following:

- SampleService_ENDPOINT.WSDL (SOAP WSDL – exposed as service)
- SampleService_ENDPOINT.WADL (REST WADL – exposed as service)

See Creating a BPEL Process to Consume the Adapter.

Creating a BPEL Process to Consume the Adapter

Within the adapter testing application, a BPEL process receives data from the adapter and replies with a response.

In this case, response values are hard-coded to send the response back by way of the adapter service.

1. Right click the Component Panel and insert a BPEL process.
2. Configure the BPEL process.

3. Click OK.

4. Wire the SampleService to the BPEL process.
5. Click OK.

6. Double click the BPEL process to open it.

7. Drag and drop a Receive activity to receive the request from SampleService.

8. Drag and drop a Reply activity to send the response back to SampleService.


10. Edit the Receive activity. Check **Create Instance** to create a new instance for every request. Create the input variable.
11. Click **OK**.

12. Wire the Reply activity to SampleService.

13. Edit the Reply activity to specify values such as the output variable. The PortType and Operation are populated automatically.
14. Click OK.

15. Drag and drop an Assign activity between the Receive and Reply activities.

16. Double click the Assign activity to edit the response values. Expand processResponseMessage (the Reply variable), right click on the PartyId response variable, choose Select Expression, and assign a static value. Assign a static value to the OrganizationName response variable.
17. Save the project.

See Deploying the Application that Tests the Inbound Adapter.

Deploying the Application that Tests the Inbound Adapter

You must deploy the application that uses the adapter before you can test the adapter.

1. Configure the Weblogic SOA server and start the server instance.
   The server is named IntegratedWeblogicServer in this example.

2. Right click Project > Deployment > New Deployment Profile...

3. For the Profile Type, select SOA-SAR File, and type a Deployment Profile Name.

4. Accept the default values for the other options and click OK.

5. After the IntegratedWeblogicServer starts, Right click Project > Deployment > Profile-Name.

6. Select Deploy to Application Server and click Next.

7. Use default values or change the Revision ID.

8. Make sure to select this option: Overwrite any existing composites with the same revision ID. This redeploys to the same version of a project.


10. Make sure the server is running and that the project is deployed in the correct partition.

11. Click Next to see a summary of your selections.

12. Click Finish.

See Testing the Inbound Adapter Using the EM Console.

Testing the Inbound Adapter Using the EM Console

After the application that uses the adapter is deployed, you can run it in the Enterprise Manager (EM) Console with the help of a SOAP application.

1. Open the EM Console after successful deployment.
   The URL is http://hostname:port/em.

2. Enter your credentials and log into the EM Console.

3. Expand the soa-infra node and expand the default domain.
4. A JCA adapter-based service is created under the Services section.

   You can’t test JCA-based services directly, so a wrapper class exposes the JCA-based service as a SOAP-based service (web service).

5. Look at the Weblogic console logs to get the service reference URI.

   Content from web.xml is displayed after successful deployment of the SOA project. For example:

   ```xml
   <?xml version="1.0" encoding="UTF-8"?>
   <jav:web-app xmlns:jav="http://java.sun.com/xml/ns/javaee" version="3.0">
     <jav:servlet>
       <jav:servlet-name>ICS_EndPoint</jav:servlet-name>
       <jav:servlet-class>oracle.cloud.connector.impl.soap.HttpTransportServlet</jav:servlet-class>
       <jav:init-param>
         <jav:param-name>service-ref</jav:param-name>
         <jav:param-value>integration/flowsvc/sample/default/SampleCloudAdapterInboundTest/SampleService/v1.0</jav:param-value>
       </jav:init-param>
     </jav:servlet>
     <jav:servlet-mapping>
       <jav:servlet-name>ICS_EndPoint</jav:servlet-name>
       <jav:url-pattern>/</jav:url-pattern>
     </jav:servlet-mapping>
     <jav:security-constraint>
       <jav:web-resource-collection>
         <jav:web-resource-name>AllEndPoints</jav:web-resource-name>
         <jav:url-pattern>/</jav:url-pattern>
       </jav:web-resource-collection>
       <jav:user-data-constraint>
         <jav:description>SSL Required</jav:description>
         <jav:transport-guarantee>CONFIDENTIAL</jav:transport-guarantee>
       </jav:user-data-constraint>
     </jav:security-constraint>
     <jav:login-config>
       <jav:auth-method>BASIC_PLAIN</jav:auth-method>
     </jav:login-config>
   </jav:web-app>
   ```

6. Create a new URL based on the service reference URI.

   The URL format is:

   https://hostname:ssl-port/integration/flowsvc/adapter-ID/weblogic-domain/client-project-name/service-name/project-version/?WSDL
In this case, the WSDL URL is:

https://hostname:ssl-port/integration/flowsvc/sample/default/SampleCloudAdapterInboundTest/SampleService/v1.0/?WSDL

7. Create a new SOAP project with this URL.

8. Double click Request1 to open the default request XML file.

9. Submit the request without any changes as a BPEL process. Do not include any logic with the payload.

You receive a response such as the following:
This chapter describes advanced features of the Cloud Adapter SDK, including batch and bulk operations and metadata downloading and caching.

**Topics**

- Design-Time Framework Description
- Batch and Bulk Operations
- Response Mapping
- Queries and DSLs
- Edit Mode
- Metadata Downloading and Caching
- Error Handling Support for REST-Based Adapters
- Runtime Framework Execution Flow

**Design-Time Framework Description**

This section shows a diagram of the design-time classes and interfaces.

The following class diagram shows the design-time classes and interfaces. These are described in further detail in the following sections according to the functional components.
Batch and Bulk Operations

A bulk operation is a single-target operation that can take a heterogeneous list of business objects. A batch operation includes multiple target operations that each can take a homogeneous or heterogeneous list of business objects.

Cloud Applications can support multiple ways of receiving large and heterogeneous data sets in a request. Furthermore, the term bulk or batch might be applied to such a process despite differences among application providers. This specification treats bulk and batch as two distinct features, and the terms are not interchangeable.

Bulk Operations

A bulk operation in the design-time (DT) framework is a single-target operation that can take a heterogeneous list of business objects. For example, a CRUD API might create the business objects Employer, Manager, and Address. A plugin that supports bulk operations permits multiple business objects of different types to be passed as request objects for generating an operation in the integration-friendly WSDL.

The following diagram shows the execution flow for modeling and generating bulk operations with the DT adapter.
One target operation and its business objects are included in the request. With this information as input, the generator produces an integrated WSDL with a single operation that references a single request element. Business objects are represented as complex types in the integration-friendly schema.

The following non-normative code example shows how a bulk operation might look in the generated integration WSDL.

```xml
<xs:complexType name="Contact">
</xs:complexType>
<xs:complexType name="PurchaseOrder">
</xs:complexType>
<xs:complexType name="CreateContactAndPurchaseOrder">
<xs:sequence>
<xs:element name="Contact" type="tns:Contact" minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="PurchaseOrder" type="tns:PurchaseOrder" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:element name="CreateContactAndPurchaseOrderRequest" type="tns:CreateContactAndPurchaseOrder"/>
<xs:element name="CreateContactAndPurchaseOrderResponse" type="tns:CreateContactAndPurchaseOrder"/>
<wsdl:message name="CreateContactAndPurchaseOrderRequestMessage">
<wsdl:part name="parameters" element="types:CreateContactAndPurchaseOrderRequest"/>
</wsdl:message>
<wsdl:message name="CreateContactAndPurchaseOrderResponseMessage">
<wsdl:part name="parameters" element="types:CreateContactAndPurchaseOrderResponse"/>
</wsdl:message>
<wsdl:portType name="ApplicationPortType">
<wsdl:operation name="CreateContactAndPurchaseOrder">
<wsdl:input message="tns:CreateContactAndPurchaseOrderRequestMessage"/>
<wsdl:output message="tns:CreateContactAndPurchaseOrderResponseMessage"/>
</wsdl:operation>
</wsdl:portType>
```

In the example above, two complex schema types represent the business objects Contact and PurchaseOrder. The operation CreateContactAndPurchaseOrder can take any number of either of these objects in the request. (In this example they must follow a particular order, but the DT framework allows you to change the order.) The default response generated for operations simply echoes the complex types sent in the request. This is customizable using the API in the DT framework.

**Batch Operations**

A batch operation in the DT framework includes multiple target operations that each can take a homogeneous or heterogeneous list of business objects. Although batch operations appear to be multiple operations, only a single operation is generated in the integration-friendly WSDL. Each target operation from the application model is converted into a request object in the integration-friendly schema. This way, the single WSDL operation can batch all tasks into one operation. This behavior is the default paradigm for batch operations in Cloud application adapters. However, batch may not
be supported by some adapter plugins, or there may be a divergence from this paradigm.

The following diagram shows the execution flow for modeling and generating batch operations with the DT adapter.

Multiple target operations and multiple business objects for each operation are included in the request. Operations can share objects. The generator takes this data and produces an integration-friendly WSDL with a single operation. First, a wrapper complex type is created for each selected target operation. Then a single request complex type is created as a container for the wrappers. The following non-normative example shows how a batch operation might look in a generated integration WSDL:

```xml
<!-- schema types -->
<xs:complexType name="Contact">
<!-- type contents -->
</xs:complexType>
<xs:complexType name="PurchaseOrder">
<!-- type contents -->
</xs:complexType>
<xs:complexType name="Invoice">
<!-- type contents -->
</xs:complexType>

<!-- Create operation type wrappers -->
<xs:complexType name="CreateContactAndPurchaseOrder">
  <xs:sequence>
    <xs:element name="Contact" type="tns:Contact" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="PurchaseOrder" type="tns:PurchaseOrder" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<!-- Create Batch Item Wrapper -->
<xs:complexType name="BatchItem">
  <xs:choice>
    <xs:element name="Contact" type="tns:Contact" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Invoice" type="tns:Invoice" minOccurs="0" maxOccurs="unbounded"/>
  </xs:choice>
</xs:complexType>
```

Batch and Bulk Operations
In the above example, two distinct business CRUD operations, Create and Update, are combined into one batch operation. A container request element combines elements from each business operation. The `xs:choice` element allows the operations to be mixed. This is handy when interleaving the business operation is necessary, for example, when updating a contact and then creating a new invoice for it.

**Response Mapping**

You can override the default response mapping if necessary.

By default, if no mapping information is received for a target operation’s response, the runtime generator copies the existing response information, if any, from the target operation in the Cloud Application Model. This includes all elements and types related to the response.

In some cases this may not be desirable. The `OperationMapping` interface defines a `setResponseMappings(List<TypeMapping>)` operation that overrides the default mapping response behavior. For example, for some CRUD operations, it might make sense to copy the business objects selected for the request mappings into the response mappings (or use the same `TypeMapping` list).

**Queries and DSLs**

Some cloud applications allow customers to run SQL-like queries on their databases. The queries are typically written in a DSL (Domain Specific Language) modelled after
standard SQL. The design-time API provides the `QueryValidator` interface, which can validate user-provided query strings.

The validator is invoked before a user saves an adapter with query information. This interface is completely optional and only applies to cloud application plugins that have a query language.

By default, the runtime generator saves the query string in the Cloud Application Configuration file. Plugins can override this process and save the query string elsewhere. Users often need to pass variables to a query string at runtime. Variables in a query string are known as **bind variables**. The design-time API provides a bind variable syntax that ensures variable substitution at runtime. The following non-normative query string shows an example of the bind variable syntax.

```
Select address.street, address.state from Contact c where c.name = :name and c.organization.name = :orgName
```

The query string above defines two bind variables, `name` and `orgName`. To assign values to these variables, the design-time framework provides classes capable of generating an XML schema complex type that represents all parameters to a query. An example of the generated complex type is shown below:

```
<complexType name="QueryParameters">
  <sequence>
    <element name="name" type="xs:string"/>
    <element name="orgName" type="xs:string"/>
  </sequence>
</complexType>
```

In the snippet above, an element was created for each bind variable. Users can now map data to the specific bind variable passed to the query at runtime.

**Edit Mode** Applications such as JDeveloper use the Cloud Adapter design-time framework to add external references to a composite project. Wizard users often modify these references after adding them. The state in which a reference is modified is known as **edit mode** in the design-time framework.

The JDeveloper wizard is in edit mode if it meets the following requirements when relaunched:

- The same business objects used to create the reference are selected.
- The same target operation used to create the reference is selected.
- With the exception of security related properties, the same connection information used to create the reference is included. Because modifying a reference can have adverse effects on the existing project and generation process, a few restrictions exist for edit mode:
  - The name of an existing operation cannot be changed.
  - Business objects can be added or removed but existing ones cannot be modified.

In edit mode, the wizard passes the previously generated WSDL and Cloud Application Configuration file as input to the `RuntimeMetadataUtil` class, which produces a list of `OperationMapping` instances. These `OperationMapping` instances are the exact mappings passed to the `RuntimeMetadataGenerator` when generating runtime artifacts. Edit mode reuses the previously generated WSDL and
Cloud Application Configuration file and preserves comments or minor changes made to those files.

**Metadata Downloading and Caching**

You can cache and download metadata to improve adapter performance.

For some ICS adapters (called connections), downloading and processing of application metadata can be a network- and processor-intensive task. In addition, because adapters are shared, the loading delay for a particular adapter is constant. For example, a delay as long as 30 seconds might occur every time a wizard user drags a particular adapter onto a source or target in ICS. Editing an adapter might involve delays as well.

In JDeveloper, the loading delay is slightly shorter and less of an issue because adapters are cached and not shared.

This delay has two sources:

- Downloading remote metadata artifacts. ICS Connections rely on SaaS application metadata, which typically comes in two forms: static artifacts such as WSDL and JSON files, and dynamic artifacts such as metadata in XML format. Downloading of static artifacts is relatively quick, but downloading of dynamic artifacts can take longer because of an application layer. For example, Oracle RightNow has a web service application to service all metadata calls.

- Parsing downloaded artifacts. After downloading, artifacts must be parsed. The adapter plugin to which the ICS Connection belongs parses the artifacts. Parsing of standard static artifacts is relatively quick, but parsing of custom metadata can be time-consuming. Some metadata requires many xpath evaluations, for which processing time grows in proportion to the document size.

Downloading and parsing of artifacts produces the Cloud Application Model (CAM). The CAM selects objects, operations, and header properties, and generates runtime artifacts. The CAM can change over time. For example, to save bandwidth, some plugins only download object names. Other details, such as field names, are downloaded during runtime artifact generation. Without caching, the CAM would be created every time a connection is dragged onto a source or target or opened for editing.

The metadata downloading and caching feature ensures that connection metadata is downloaded and parsed only when the connection is saved. This creates a one-to-one relationship between a connection and its metadata. This feature is optional. Only plugins that can use the performance benefits need to implement the required interfaces. Other plugins are not affected.

The following diagram shows the details of the metadata downloading and caching feature.
Here is a more detailed description of the process:

1. The process begins when the wizard user saves a connection.

2. The connection properties are checked for values. Empty connection properties mean no caching is required and no download is triggered.

3. Valid connection properties trigger the metadata download. Part of the download process (not shown in the diagram) is to check whether the plugin supports metadata downloading.

4. If a download is already in progress for this connection, then no additional download occurs. The save is marked as complete.

5. If a CAM is already cached, the digest of connection properties is checked for changes. Plugins that implement the caching API can optionally mark signature properties. Changes to signature properties indicate that the cached data is invalid. For example, in the RightNow plugin, the WSDL URL, username, and password are signature properties.

6. If the digest has not changed, the cached data might still need to be refreshed. Plugins that implement the caching API can optionally implement the refreshRequired method, which receives the lastModified date of the metadata. If this method returns true, the cache is refreshed.

Note: If the download of cached data fails, the save does NOT fail. Downloading of metadata is not mandatory, and even if it fails or does not exist, the connection still works, though most likely more slowly.

Implementation Details for Adapter Plugins
To support downloading and caching of metadata in a plugin, follow these steps:

1. The main class that extends AbstractCloudApplicationAdapter must also implement oracle.tip.tools.ide.adapters.cloud.api.model.HandlesBackgroundMetadataDownload. This interface has one method:
   getMetadataDownloadHandler().

   ```java
   public class RightNowCloudAdapter extends AbstractCloudApplicationAdapter
   implements HandlesBackgroundMetadataDownload {
   ```

2. The oracle.tip.tools.ide.adapters.cloud.api.model.MetadataDownloadHandler returned from the getMetadataDownloadHandler() method must be implemented.

   **Tip:** have your implementation of AbstractMetadataBrowser implement this interface and return a new instance of the metadata browser. The browser should already contain details for downloading and parsing metadata, so you don’t need to write a new class.

   ```java
   @Override
   public MetadataDownloadHandler getMetadataDownloadHandler(AdapterPluginContext context) {
       return new SampleMetadataBrowser((CloudConnection)context.getContextObject(AdapterConstants.CONNECTION),context);
   }
   ```

The following diagram shows the interfaces and classes required to implement caching:

---

**The HandlesBackgroundMetadataDownload Interface**

This interface must be implemented by the main adapter class of your plugin. The main adapter class is the class that extends oracle.tip.tools.ide.adapters.cloud.impl.plugin.1AbstractCloudApplicationAdapter. An adapter plugin that implements this interface is considered to be supporting the downloading of metadata in a background thread.

**Note:** The adapter is not responsible for managing threading. The design-time framework uses a background thread to invoke the main method of this interface,
getMetadataDownloadHandler, which must return an implementation of MetadataDownloadHandler.

```java
public class SampleMetadataBrowser extends AbstractMetadataBrowser implements MetadataDownloadHandler {
  // Constructor
  public SampleMetadataBrowser(CloudConnection connection, AdapterPluginContext context) {
    super(connection, context);
    String ns = null;
    CloudApplicationModel model = getModel();
    if (model != null) {
      // Try to use the cached WSDL to get the namespace
      Definition def = (Definition) model.getSourceArtifacts().get(AdapterConstants.WSDL_DEFINITION);
      if (def != null) {
        ns = def.getTargetNamespace();
      }
    }
    if (ns == null) {
      ns = SampleCloudUtil.parseWsdlNamespace(connection, context);
    }
    currentVersion = RightNowCloudUtil.getRightNowVersion(ns);
  }
}
```

**The MetadataDownloadHandler Interface**

This interface represents a handler for the plugin’s metadata downloads. The handler is invoked by a background thread that manages the cache. The main artifact of this handler is the `CloudApplicationModel`.

The `downloadMetadata()` method is called when a metadata download is required. The implementation should download and parse metadata and ultimately produce a `CloudApplicationModel`. The `CloudApplicationModel` is cached by the framework.

**Note:** The CAM does not have to be complete at this point. Incomplete parts of the CAM can be processed later using CompletionHandlers.

```java
public CloudApplicationModel downloadMetadata() throws CloudApplicationAdapterException {
  AdapterPluginContext context = getContext();
  List<MetadataParser> parsers = getMetadataParsers(context);
  for (MetadataParser parser : parsers) {
    parseMetadata(parser, context);
  }
  return (CloudApplicationModel) context.getCloudApplicationModel();
}
```

The `requiresRefresh()` method is called with the last `dateTime` of the metadata that was downloaded. The plugin should determine whether the current metadata is still fresh or whether a new download is necessary. Returning `true` causes new metadata to be downloaded, while `false` keeps the current metadata.

```java
@Override
public boolean requiresRefresh(long lastModified) throws CloudApplicationAdapterException {
  boolean doRefresh = true;
  // Implementation...
  return doRefresh;
}
```
try {
// ping ds and record last modified time.
SampleCloudConnection conn = (SampleCloudConnection)getContext().getContextObject(AdapterConstants.CONNECTION);
PingStatus pingStatus = conn.ping();
if (pingStatus.isSuccess()){
    long currentModified = getCurrentModified(getContext());
    doRefresh = currentModified > lastModified;
} else {
    doRefresh = false;
}
} catch (Exception e) {
    doRefresh = true;
}
return doRefresh;
}

The `signatureKeys()` method returns a set of properties that form the signature of the cached metadata. When this signature changes, the cache is considered invalid. For example, an adapter plugin might consider the WSDL URL, username, and password properties to be signature keys. If any of these properties change in a connection, the currently cached metadata is considered invalid and new metadata is downloaded.

```java
@override
public Set<String> signatureKeys() {
    // if any of these properties change then the cache should be invalidated
    Set<String> keys = new HashSet<String>();
    keys.add(AdapterConstants.SOURCE_WSDL_LOCATION);
    keys.add(AdapterConstants.USERNAME);
    keys.add(AdapterConstants.PASSWORD);
    return keys;
}
```

### Error Handling Support for REST-Based Adapters

This section describes what a REST-based adapter plugin must implement to support error handling.

#### Design Time

At design time, the fault pipeline appears only if the WSDL operation has a fault message. Therefore, adapters should include a `faultType` in their WSDL operation for propagating faults. The Adapter SDK defines faults for adapter plugins and provides helpers to include these faults as part of the adapter WSDL. Faults can be added to the `RESTOperation` node by calling the following helper class:

```java
// set fault node
RESTHelper.setFaultOnOperationNode((RESTCloudOperationNodeImpl) opNode, context);
```

#### Runtime

At runtime, an outbound adapter should propagate faults. An inbound adapter should handle propagated faults to return meaningful error responses, with a valid HTTP error code and a helpful error message.

#### Outbound Adapter Fault Handling

Adapter plugins extend the `CloudMessageHandler` class. You can override the `handleRequest`, `Response`, and `Error` handlers to do adapter-specific processing. The Adapter SDK provides a `RESTFaultGenerator` that can propagate any failures.
that occur during each processing step. Use the following methods to report meaningful errors.

The `handleRequestMessage()` method performs header and payload processing:

```java
try{
    // Header processing logic
} catch(Exception e){RESTFaultGenerator.RequestHandler.headerProcessingException(e);
}
try{
    // Payload processing logic
} catch(Exception e){RESTFaultGenerator.RequestHandler.payloadProcessingException(e);
}
```

The `handleResponseMessage()` method performs payload processing:

```java
try{
    // Payload processing logic
} catch(Exception e){RESTFaultGenerator.ResponseHandler.payloadProcessingException(e);
}
```

In addition, any error that occurs during invocation or any error response from a target service is automatically handled by the Adapter SDK.

### Runtime Framework Execution Flow

This section illustrates and describes the runtime framework execution flow. The following diagram shows the basic flow of the runtime framework.
In the RT framework, each application has a corresponding RT plugin. Each RT plugin must conform to the Cloud Application Adapter API, which provides the interfaces to be implemented. The single entry point for SOA and OSB processes is the Cloud Application Adapter. This entry point is always present in deployments with a Cloud
Application Configuration file. The Cloud Application Configuration file contains a location attribute that identifies the Cloud Application Adapter entry point:

```xml
<connection-factory location="cloud/CloudAdapter"/>
```

The adapter is only loaded once. Upon invocation, the following execution flow is followed:

1. The Cloud Application Adapter RT looks up the RT plugin instance by checking the `managedConnectionFactoryClassName` attribute value. The instance is only loaded once.

2. If the RT instance was not previously loaded, then a new instance is created from the plugin’s implementation of `oracle.cloud.connector.api.CloudConnectionFactory`. In addition to this instantiation, various connection level properties are set.

3. Once the plugin instance has been retrieved, the target cloud application operation is invoked. The RT Framework gets an instance of `oracle.cloud.connector.api.CloudOperation` and calls the invoke method, passing in the request payload.

4. During the invoke call, the plugin’s Operation instance performs all the necessary transformations on the request payload and constructs a payload to send to the cloud application. The RT framework provides special invoker interfaces that can send SOAP, JSON, or XML messages to REST or SOAP endpoints.

5. If invocation of the cloud application was successful, the plugin performs any necessary transformations on the message and returns an instance of `oracle.cloud.connector.api.CloudMessage` containing the actual response.

6. If invocation of the cloud application failed, the plugin handles the exception and throws an instance of `oracle.cloud.connector.api.CloudConnectorFault` containing the actual fault details.
This appendix contains reference information, including troubleshooting and JDeveloper design tips, a complete list of classes and interfaces, and UI field object information.

**Topics**

- Adapter Troubleshooting
- JDeveloper Wizard Design Recommendations
- Unified UI Framework Classes
- Unified UI Field Object Names and Descriptions

### Adapter Troubleshooting

This section describes common problems and solutions related to installation and configuration of Oracle adapters.

**Cloud Adapter 12.1.3 Patch installation was unsuccessful**

**Solution:** Refer to the Cloud Adapter SDK Installation Guide.

**Can’t see the custom adapter even after registration**

**Solution:**

- Validate and ensure that the extension ID (in extension.xml) and Bundle-Name (in MANIFEST.MF) match and are unique.
- Validate and ensure that the adapterPluginID (in cloud-adapter.xml) and ADAPTER_NAME (in SampleCloudAdapterConstants) match and are unique.

**Can’t create an External Reference or Exposed Service**

An error pops up when a wizard user drags and drops a newly built (samplecloudadapter) adapter onto the External Reference or Exposed Services pane.

**Solution:**

- Validate and ensure that the implementation-class property value in the extension.xml file is equivalent to the SampleCloudScaEndpointImpl class in the wizard package.
- For ICS or Service Bus, validate and ensure that the adapterFactory property value in cloud-adapter.xml is equivalent to the Factory (SampleCloudAdapterFactory) class defined in the plugin package.
• Validate that the PageID in the UIBinding class matches the PageID in the ICloudAdapterPage.

• // Adding request config page to wizard in UIBinding class
  editPages.put(SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REQ_CONFIG_PAGE_ID,
   new SampleInboundRequestConfigPage(context));

  // SampleInboundRequestConfigPage uses
  public String getPageId(){
   return SampleCloudAdapterConstants.SAMPLE_CLOUD_APP_INBOUND_REQ_CONFIG_PAGE_ID;
  }

Ping is not successful
Solution:
• Check if any proxy settings are applied. By default, JDeveloper has System Proxy Settings.
• Verify that the application service URL is up and running.
• Ensure that the ping method (getAccount) and method namespace match the target service.

Runtime invocation fails
Solution:
• Ensure that the META-INF.MF file has all the dependency JAR files.
• Validate that the CONNECTION_FACTORY_CLASS constant value matches the SampleCloudAppConnectionFactory class.

Runtime invocation fails because the JCA Binding invocation of the reference operation failed
The following message appears:
Can't find dispatch method for {<NAMESPACE>}<OPERATION_NAME>

Solution: Print the request payload after the root element normalization.
Go to SampleCloudAdapterMessageHandler.
Operation: processRequestHeaders
Statement: this.printXML(doc);

**JDeveloper Wizard Design Recommendations**
This section lists wizard design recommendations for running an adapter in JDeveloper.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single invocation of an adapter wizard should generate one integration WSDL file and one JCA file.</td>
<td>Under SOA (and OSB) in JDeveloper, only a single composite service or reference is generated.</td>
</tr>
</tbody>
</table>
Unified UI Framework Classes

This section lists the classes, interfaces, and methods in the Unified UI (or Adapter Wizard) Framework.

Throughout this section, AUUIF is an acronym for Adapter Unified UI Framework.

**CloudAdapterUIProvider Abstract Class**

This abstract class must be extended by adapters, and all methods must be implemented. This class creates the UIBinding corresponding to the adapter. Also, the consumer inputs and the locale are passed to UIBinding. UIBinding uses this info to generate pages and fields in the wizard.

**Methods**

**ICloudAdapterUIBinding getCloudAdapterUIBinding()**

This method creates a new instance of the adapter-specific UIBinding (subclass of ICloudAdapterUIBinding) and returns it. The client’s locale and the consumer’s input are passed to the UIBinding object from here.

**Code Example:**

```java
uiBinding = new SampleCloudUIBinding(cloudAdapterFilter, locale);
```

**String getLocalizedAdapterType() — Localized String: YES**

This method returns the adapter type, such as RightNow, SalesForce, and so on. This string is part of the wizard header and must be localized.

**ICloudAdapterUIBinding Interface**

You can either implement this or extend AbstractCloudAdapterUIBinding.

The implementation class of ICloudAdapterUIBinding:

- Creates the DefaultAdapterPluginContext, an adapter-specific context for creating artifacts. This context object is also passed to all pages so values can be passed from one page to another.

- Creates the actual adapterProvider (subclass of AbstractCloudApplicationAdapter), and sets this in the context created in the previous step.
• Allows you to set any other object used in pages in this context. For example, the locale and adapterFilter are passed to this class.

This interface has APIs to get the list of pages, get the UI fields for each page, perform business logic validation, delegate page actions to respective pages, and generate artifacts. This interface handles interaction between the wizard and the actual adapter pages. This class should specify the following constructor.

**Constructor:**

```
SampleCloudUIBinding<CloudAdapterFilter cloudAdapterFilter, Locale locale>
```

Initializes the adapter and the adapter-specific context.

**Code Example:**

```java
this.context = new DefaultAdapterPluginContext(SampleCloudAdapterScaEndPoint.JCA_ADAPTER_SUBTYPE_ID, referenceName, null);
AbstractCloudApplicationAdapter cloudApplicationAdapter = new SampleCloudAdapter(this.context);
this.context.setContextObject(CloudAdapterConstants.APPLICATION_ADAPTER, cloudApplicationAdapter);
this.context.setContextObject(CloudAdapterConstants.UI_CLOUD_ADAPTER_FILTER, cloudAdapterFilter);
```

**Methods:**

**LinkedHashMap<String, ICloudAdapterPage> getEditPages()**

This method returns the list of pages in the wizard. The key in the map is *pageId* and the value is an instance of *ICloudAdapterPage*.

The sequence of pages in the map determines the sequence in the wizard.

**Code Example:**

```java
LinkedHashMap<String, ICloudAdapterPage> editPages = new LinkedHashMap<String, ICloudAdapterPage>();
editPages.put(CloudAdapterConstants.WELCOME_PAGE_ID, new SampleCloudWelcomePage(context));
editPages.put(CloudAdapterConstants.OPERATIONS_PAGE_ID, new SampleCloudOperationsPage(context));
```

**CloudAdapterPageState getUpdatedEditPages()**

This method is called whenever events are generated in the wizard and the components generating these events have *eventListener* set on them. Examples are ButtonClick, CheckBox value change, SelectBox value change, and so on. This method forwards the request to the corresponding page, which handles the event. See *ICloudAdapterPage* for more details.

**LinkedHashMap<String, UIError[]> validatePage()**

This method is called when navigating to next page. The request is forwarded to the corresponding page. See *ICloudAdapterPage* for more details.

**CloudAdapterPageState getNextPage()**

This method is called if *validatePage()* doesn’t return errors. The request is forwarded to corresponding page’s *updateBackendModel()*. See *ICloudAdapterPage* for more details.
CloudAdapterPageState getPreviousPage() — This is not used currently.

HashMap<String, byte[]> generateMetadataArtifacts()

This method is called when the Done button in the wizard is clicked. The logic to generate all the artifacts, such as JCA, WSDL, and WSDL Operation, must be implemented. These artifacts are set in a map and returned to AUUIF.

String getWizardTitle() — Localized String: YES

Returns the wizard title.

getAdapterConfiguration() — This method is not used currently.

AbstractCloudAdapterUIBinding Class

This class implements ICloudAdapterUIBinding and provides the default implementation for methods common to most adapters. You can decide based on your requirements whether this class should be extended or ICloudAdapterUIBinding should be implemented.

ICloudAdapterPage Interface

This interface must be implemented, but in most cases it is already implemented indirectly. The standard page classes, such as CloudAdapterConnectionPage, extend AbstractCloudAdapterPage, which implements ICloudAdapterPage. If you don’t use the standard page classes or extend AbstractCloudAdapterPage, you must implement ICloudAdapterPage.

This interface builds a new page, specifying the fields required on the page, the order of the fields, business logic validation, and so on.

The following figure shows how the wizard gets the values to display from the adapter.

Methods:

getPageId() — Localized String: NO

Returns the unique pageId associated with this wizard page. The pageId is used internally to locate the page. This is not exposed to wizard users.
getPageName() — Localized String: YES
Returns the pageName. The pageName is displayed in the list of pages.

getPageTitle() — Localized String: YES
Returns the pageTitle. The pageTitle is displayed at the top of the page.

getWelcomeText() — Localized String: YES
Returns the text which briefly describes the page.

getPageEditFields()
Returns the list of UI fields to be displayed on the page. The order of fields in the list determines the order on the page.

This method is called only when fresh UI fields are needed, usually when the page is loaded for the first time. These fields are cached at the UI layer, and the cached fields are used afterward.

These UI fields are from the UI Objects SDK, which contains a simple Java-based set of components you can use to build the page. The Adapter Wizard Framework (AWF) converts these fields to ADF fields before rendering the page.

**Code Example:**

```java
// Creating a simple textbox using UIObjects SDK’s edit fields.

LinkedList<EditField> fields = new LinkedList<EditField>();
Locale locale = CloudAdapterUtils.getLocale(adapterPluginContext);
TextBoxObject textBoxObject = UIFactory.createTextBox("");
fields.add(UIFactory.createEditField(CloudAdapterConstants.REFERENCE_NAME,
                                    CloudAdapterText.getString(locale,
                                    CloudAdapterText.WELCOME_PAGE_REFERENCE_NAME),
                                    CloudAdapterText.getString(locale,
                                    CloudAdapterText.WELCOME_PAGE_REFERENCE_DESC),
                                    true, textBoxObject));
return fields;
```

getChildrenEditPages()

A page can have child pages. Scenarios in which child pages may be required are:

- Pop-ups are not supported.
- Tabs are not supported.
- If a value in one field requires the display of several more fields, it might make sense to display the additional fields in a separate page.

For example, in the CloudOperationsPage, a separate childPage can be added for testing an ROQL query.

Whenever a child page is loaded, it replaces the parent page in the wizard. That is, the parent and child pages share the same space in the wizard. When the child page is displayed, the train component, which lists all the wizard pages, is disabled. Navigation to wizard pages other than the parent is not possible from child page. Full navigation is reenabled upon return to the parent page.

getUpdatedEditPages()

This method is called whenever wizard components that generate events have eventListener set. Examples are ButtonClick, CheckBox value change,
SelectBox value change, and so on. The method implementation must meet these requirements:

- The fieldName parameter tells which field on the page generated the event, and the event should be handled accordingly.
- If there are any changes to the sequence of wizard pages or additions or deletions to wizard pagesList based on the event, modify the wizardPages parameter and send the modified pagesList back in the return object.
- If there are any additions or deletions to the page's fieldsList based on the event, the modified fieldsList should be sent back in the return object.
- Set refresh to true in the return object if the wizard needs to be refreshed. This may be required if there are additions or deletions to wizard pages.
- Set the modified wizard pages list, modified fields list, and refresh value in CloudAdapterPageState and return it. Refer to the CloudAdapterPageState description below.

validatePage()

This method, which performs business logic validation, is called just before leaving a page and before calling updateEditPage(). If validation fails, the corresponding error should be set in errorsList and returned.

Code Example:

```java
LinkedHashMap<String, UIError[]> errors = new LinkedHashMap<String, UIError[]>();
UIError[] errorArray = new UIError[errorList.size()];
errorArray[0]=new UIError(fieldsName1,err1);
errorArray[1]=new UIError(fieldsName2,err2);
...
errors.put(getPageId(),errorArray);
```

The wizard uses this list to show and highlight the error field.

updateBackEndModel()

This method updates field values in the back-end model. This method is called just before leaving the page.

Code Example:

```java
@Override
cublic CloudAdapterPageState updateBackEndModel(LinkedHashMap<String, ICloudAdapterPage> wizardPages, LinkedList<EditField> currentPageFields) {

    Properties props = new Properties();
    // captures current edit fields...
    Map<String, UIObject> map = EditField.getObjectMap(currentPageFields.toArray(new EditField[currentPageFields.size()]));
    CloudAdapterPageState state = new CloudAdapterPageState(false, wizardPages, currentPageFields);

    // Set BusinessObject Name and CloudOperation Name to contextObject to display in summary page.
    UIObject uiobject = (UIObject)map.get(CloudAdapterConstants.CLOUD_BIZ_OBJ);
    String bizObjName = ((IShuttleObject)uiobject).getSelectedValues().get(0);

    this.adapterPluginContext.setContextObject(CloudAdapterConstants.CLOUD_BIZ_OBJ,
```
String cloudOpName = UIFactory.getStringValue(map, CloudAdapterConstants.CLOUD_OPERATION);
this.adapterPluginContext.setContextObject(CloudAdapterConstants.CLOUD_OPERATION, cloudOpName);

CloudMetadataBrowser browser = CloudAdapterUtils.getMetadataBrowser(adapterPluginContext);
// This is a convenience class that will add objects/operations to the //TransformationModelBuilder. You can of course choose to do this yourself. //But if this works for you then you should just use it.

CloudAdapterUtils.setOperationMappingsForGenerator(adapterPluginContext, map, browser, props, getSelectedOperation());

return state;

AbstractCloudAdapterPage Class
This class implements ICloudAdapterPage and provides a default implementation of common methods. You can decide, based on requirements, whether to extend this class or implement ICloudAdapterPage.

Note: Using this class is recommended over implementing ICloudAdapterPage.

CloudAdapterPageState Class
Methods such as getUpdatedEditPages() and updateEditPage() return a CloudAdapterPageState object. These methods can change the sequence in which pages are rendered, add more pages to the pagesList, or delete pages from the pagesList. Similarly, fields can be added or deleted or their sequence can be modified. Set the modified pagesList and editFieldsList in the CloudAdapterPageState object. For navigation from a parent page to a child page and back, set the childPageId in this object.

Use refreshWizard if the complete wizard needs to be refreshed. Use nextPageId if the next page is different from the initial page sequence. Use refreshParentPage if the parent page needs to be refreshed after returning from a child page.

Methods:
isRefreshChildPage() and setRefreshChildPage()
This flag indicates that the child page must be refreshed with back-end model values.
isRefreshParentPage() and setRefreshParentPage()
These methods determine whether the parent page must be refreshed when navigating back from a child page. If set to true, the back-end model is updated, so that the parent page can retrieve updated values from the back-end model and display them. If set to false, the back-end model and parent page are not updated.

isNavigateToChildPage() and setNavigateToChildPage()
These methods determine whether the next page to be rendered is a childPage. If set to true, the wizard navigates to the child page. The child page must be set. If set to false, the wizard navigates to the next page in the pagesList.
isNavigateToParentPage() and setNavigateToParentPage()
These methods determine whether navigation is from a child page to a parent page. If set to true, the next page to be rendered is the parent page of current child page.

**getNextPageId() and setNextPageId()**

These methods determine which page should be rendered when Next is clicked. If the next page is different from the initial page sequence, then set this to the pageId of the next page. If this value is not set, the default next page from the pagesList is rendered.

**getPrePageId() and setPrePageId()**

These methods determine which page should be rendered when Back is clicked. If the previous page is different from the initial page sequence, then set this to the pageId of the previous page. If this value is not set, the default previous page from the pagesList is rendered.

**getChildPageId() and setChildPageId()**

If isNavigateToChildPage() is true, these methods must specify the childPageId.

**isRefreshWizard()**

If the sequence of pages has changed or if pages are added to or deleted from the pagesList, set this flag to true to refresh the wizard’s train component.

**getUpdatedCloudAdapterPages()**

Returns the updated pagesList.

**getUpdatedPageEditFields()**

Returns updated fields for the current page.

**isConfigComplete() and setConfigComplete()**

This flag indicates whether the configuration is complete. If set to true, the Done button is enabled in the wizard.

**getUiErrors() and setUiErrors()**

These methods handle errors that occur when any method that returns CloudAdapterPageState runs. Examples are errors in validating the page or updating the back-end model. The Adapter Wizard Framework checks for errors, displays them to the wizard user, and cancels all page navigations.

**CloudAdapterFilter Class**

This class is an input to the wizard. Based on the filter value set, pages are filtered in the wizard. The consumer of the wizard, for example JDeveloper or ICS, sets the filters.

**Filters:**

- **applicationInstance** — If the consumer has a valid ApplicationInstance (a valid connection object), it can be passed to the Adapter Wizard Framework (AWF). The AWF gets information from the application instance such as the provider, the cloud adapter connection, and so on. If this is not present, the consumer must pass at least the provider ID with which the cloud adapter is registered.

- **applicationType** — For internal use.

- **applicationName** — For internal use.

- **referenceName** — The consumer can also pass the reference name. This is optional.
**providerId** — If the consumer does not have an ApplicationInstance, the providerId must be passed to determine which cloud adapter is being invoked. The providerId is a unique ID with which the cloud adapter is registered with the consumer.

**cloudAdapterArtifacts** — This filter sets the artifacts. This is mostly for when the wizard is invoked in edit (re-entrant mode). The wizard is pre-populated with the artifact values.

**inbound** — This flag indicates that the wizard is for an inbound adapter.

**username** — For internal use.

**addConnection** — Adds the connection page to the wizard pages.

**connectivityServiceFactory** — For internal use.

**cloudAdapterCallerContext** — Identifies the consumer. Valid values include ICS_WEB, SOA_JDEV, and OSB_WEB.

**adapterPluginContext** — The default adapter plugin context, which is provider-specific.

**Page Implementation Classes**

The Cloud Adapter Wizard Framework also provides implementation for four commonly used pages: the Welcome page, Connection page, Operations page, and Summary page. You can extend these classes if these pages meet your requirements with only minor modifications. Otherwise you can either extend AbstractCloudAdapterPage or implement ICloudAdapterPage directly and create pages from scratch.

Here are the classes that extend AbstractCloudAdapterPage:

- CloudAdapterWelcomePage
- CloudAdapterConnectionsPage
- CloudAdapterOperationsPage
- CloudAdapterSummaryPage

The wizard can display up to 10 pages, not including child pages.

**Unified UI Field Object Names and Descriptions**

This section describes or shows examples of Unified UI field objects.

**ButtonObject**

![Connection Test](image)

**CheckBoxObject**

![Use Proxy](image)

**ConfirmationObject**

ADF Info Message is displayed

**DynamicCompositeCell**

Used with DynamicCompositeCellRow
**DynamicCompositeCellRow**

**DynamicCompositeCellTable**
Uses DynamicCompositeCellRow and DynamicCompositeCell. DynamicCompositeCellTable acts as a container for DynamicCompositeCellRow objects.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRUD</td>
<td>Business Objects (API 1.2) commit</td>
</tr>
<tr>
<td>CRUD</td>
<td>Business Objects (API 1.2) commit</td>
</tr>
</tbody>
</table>

**DynamicTableObject**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>param1</td>
<td>string</td>
</tr>
<tr>
<td>param2</td>
<td>boolean</td>
</tr>
</tbody>
</table>

**ExpandableRowObject**
Used with ExpandableTableObject. Each ExpandableRowObject is a separate row in ExpandableTableObject.
Unified UI Field Object Names and Descriptions

**ExpandableTableObject**

**Methods**
- Java2XML_BigDecimal

**Operation**
- Java2XML_BigDecimal

**Parameters**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Converter</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigDecimal</td>
<td>arg0</td>
<td></td>
</tr>
</tbody>
</table>

**Return**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Converter</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigDecimal</td>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

- Java2XML_BigDecimalTXA
- Java2XML_BigDecimalTXA

**ExtendedDynamicTableObject**

**EndPoint Properties**

<table>
<thead>
<tr>
<th>property</th>
<th>value</th>
</tr>
</thead>
</table>

**Dynamic EndPoint Properties**

<table>
<thead>
<tr>
<th>property</th>
<th>value</th>
</tr>
</thead>
</table>

**FileBrowserObject**

**Without Textarea**

Select File [Browse...] No file selected.

**With Textarea**

Select File [Browse...] No file selected.

**GroupObject**

**HintTextAreaObject**

```plaintext
* URI
(hint: http://host:port/someservice)
```

**HintTextBoxObject**

```plaintext
* URI
(hint: http://host:port/someservice)
```

**ImageObject**

![Image](image.png)

**LabelObject**

With icon:

![Edit](edit.png)

**LinkObject**

Link with an image:

![Add](add.png)

**MessageObject**

**Error:**
The field Password is required. Please enter a value.

A password credential

**Option**

Used with SelectObject or ShuttleObject.

**PasswordObject**

```
* Password
**********
```
**PopupDialog**

![PopupDialog Image]

**ProgressBarObject**

![ProgressBarObject Image]

**QueryBuilderObject**

![QueryBuilderObject Image]

**SectionHeaderObject**

```
<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppress External Events □</td>
</tr>
</tbody>
</table>
```

**SelectBooleanRadioObject**

- None
- Basic
- Client Certificate
- Custom Authentication (See Advanced Settings)

**SelectItem**

Used with ShuttleObject.
**SelectObject**

*Horizontal (display = radio)*

- **Operation mode**
  - [ ] Single
  - [ ] Batch configuration

*Vertical (display = radio)*

- **Authentication**
  - [ ] None
  - [ ] Basic
  - [ ] Client Certificate
  - [ ] Custom Authentication (See Advanced Settings)

**Select One Choice (display = list)**

- **Dispatch Policy**
  - default

**SeparatorObject**

---

**ShuttleObject**

**Business Objects**

<table>
<thead>
<tr>
<th>Available values</th>
<th>Selected values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>Answer</td>
</tr>
<tr>
<td>AnalyticsReport</td>
<td>Contact</td>
</tr>
<tr>
<td>ChannelType</td>
<td></td>
</tr>
<tr>
<td>Country</td>
<td></td>
</tr>
<tr>
<td>GenericObject</td>
<td></td>
</tr>
<tr>
<td>Incident</td>
<td></td>
</tr>
<tr>
<td>Mailbox</td>
<td></td>
</tr>
<tr>
<td>MarketingMailbox</td>
<td></td>
</tr>
<tr>
<td>Company</td>
<td></td>
</tr>
</tbody>
</table>

*With filter:*

- **CO:**
- **ALL:**

**Select Business Objects (RightNow 1_2 API)**

- [ ] CO.Contact_CustomObj
- [ ] CO.Custom
- [ ] CO.CustomRightNow
- [ ] CO.Demo
- [ ] CO.DummyCustomObject
- [ ] CO.Quote
- [ ] CO.QuoteServiceReg
- [ ] CO.QuoteTest

**Your Selected Business Objects**

- [ ] AssetStatus
- [ ] Answer
- [ ] CO.DemoTest

---

**SimpleCell**

*Used with SimpleRowObject.*
**SimpleRowObject**
A container for SimpleCell, used with SimpleTableObject.

**SimpleTableObject**
Request Responses

<table>
<thead>
<tr>
<th>Request URI</th>
<th>Response URI</th>
<th>Service Account</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TabContainerObject**
A container for multiple TabObjects.

**TabObject**
Used with TabContainerObject. Multiple TabObject objects can be passed to TabContainerObject.

**TextAreaObject**

<table>
<thead>
<tr>
<th>Query Statement</th>
<th>select country from country</th>
</tr>
</thead>
</table>

**TextBoxObject**

<table>
<thead>
<tr>
<th>WSDL Name</th>
<th></th>
</tr>
</thead>
</table>