Oracle® Fusion Middleware

Developing Custom Technology Adapters for Oracle Fusion Middleware

11g (11.1.1.9.0)

E40792-02

February 2015

This document provides information on developing a custom Adapter using the Oracle JDeveloper, and developing an Adapter runtime component for integration with Fusion Middleware.

This document contains the following sections:

1 Primary Considerations

Oracle JCA-compliant adapters enable you to integrate your business applications, and provide a robust, lightweight, highly-scalable and standards-based integration framework for disparate applications to communicate with each other. You can create your own Custom Adapters and runtime components, based on your own requirements.

The Custom JCA Adapter wizard is a generic adapter wizard that reads and displays its interaction/activation specs, properties, and default values from a configuration file. The user employs the design time facilities of JDeveloper to use the wizard, select the specs, override the default property values, and add new properties to the adapter.

This means that the Custom JCA Adapter must obtain information from the user and use that information to create a WSDL file and a JCA file containing the interaction/activation specs and properties that are required by the runtime adapter.

1.1 Preliminary Decision: Uni-Directional or Bidirectional Adapter?

When creating a Custom Adapter, the first decision you need to make is to decide if the Custom Adapter will be uni-directional or bidirectional-will it support invocation for outbound (synchronous) operations-that is, through the JCA Common Client (CCI) API. Alternatively, should the Adapter support both outbound and inbound (asynchronous) message flow?

Once you make this decision and then create a Custom Adapter that supports outbound and inbound message flow, it is important to know that:

- To support outbound message flow, the adapter needs to define one or more implementations of `javax.resource.cci.InteractionSpec`.
- To support inbound message flow, the adapter needs to define one or more implementations of `javax.resource.spi.ActivationSpec`.

Each of these specs (Java Beans) provide metadata as name/value pairs that define a specific operation that your adapter intends to support.

For example, for an FTP adapter, the interaction spec exposes a bean property named `Directory`. When the FTP adapter is invoked, the invoker can instruct the adapter to place the payload in a specific directory, by setting the interaction spec property `Directory` - for example, by setting `FtpInteractionSpec.setDirectory("/tmp/receive")`

If the adapter supports different kinds or categories of invocations, which each require a different set of properties, you can model each invocation type by a distinct implementation of `javax.resource.cci.InteractionSpec`.

For example. the adapter might further implement `SshFtpInteractionSpec` and `SSLFtpInteractionSpec`, with the first interaction spec supporting the S-FTP protocol and the second interaction spec supporting the FTP/S protocol.

Knowing this information, and the potential use of your Adapter in terms of the operations it supports, is a helpful first step in your customization.

1.2 Purpose

Delving deeper into the Adapters, you should know that the Custom JCA Adapter wizard has several purposes:

- Customers or third party adapter providers can use the Custom Adapter Wizard "as is" to support their custom runtime adapters. They only need to supply (or extend) the Custom Adapter configuration file (`customAdapter-config.xml`).
- Customers and third party adapter providers can extend the Custom Adapter classes if they want to create a more specific adapter (for example, they can change the text to match their adapter)
- The Custom Adapter wizard is a simple example of how to develop a new adapter wizard (or to convert an old adapter wizard) using the JCA Adapter Framework and hooking into the SCAEndpoint interface. After the SOA jdev extension is installed, the Custom Adapter java source files can be found in `<JAVA_HOME>/developer/integration/adapters/samples/custom`

Before you learn about creating the Adapter design-time components, you need to understand the screens that make up the default Custom Adapter Wizard.

2 Custom Adapter Screens Typical Appearance

This section provides a walk-through of the screens the user sees when they use a typical Custom Adapter Wizard to configure an Adapter

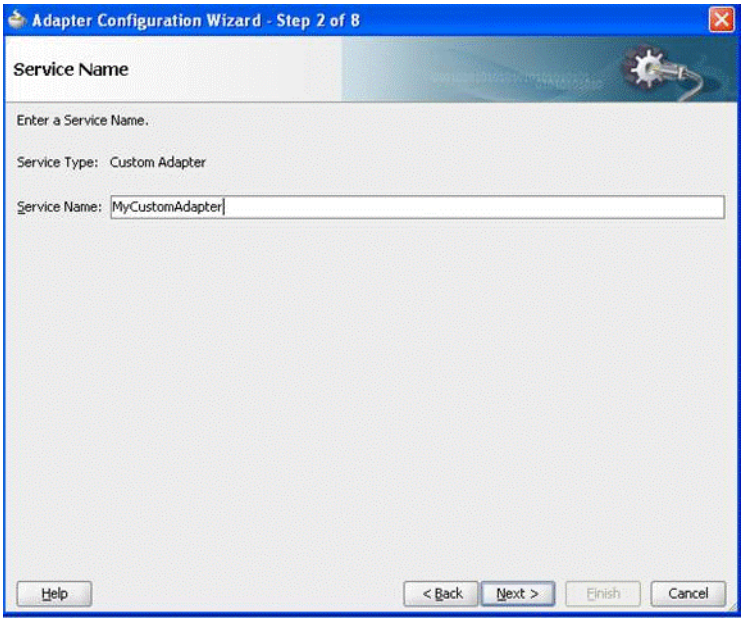
When the user drops the Custom JCA Adapter icon to the Exposed Service or External Reference swimlane in JDeveloper, Jdeveloper displays the Adapter Configuration Welcome Page.

Figure 1 The Adapter Configuration Wizard Welcome Screen

The user then selects **Next** on the Welcome page.

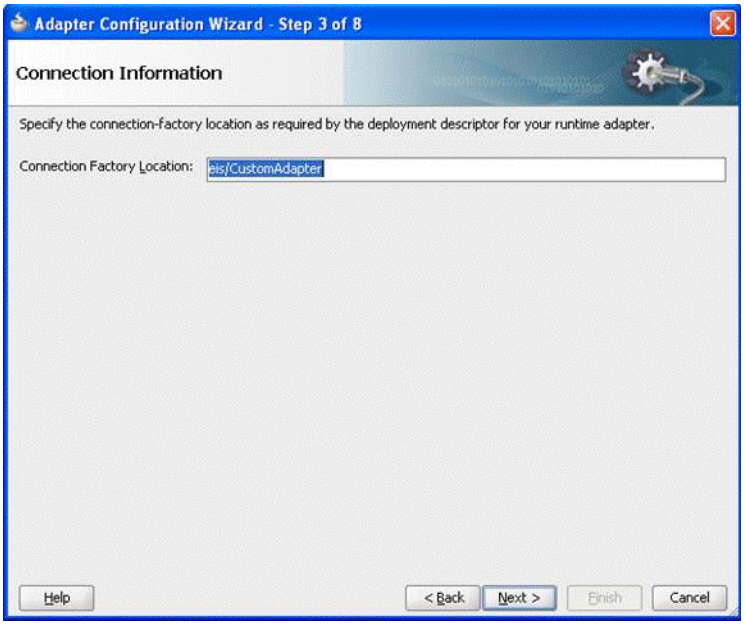
The Service Name page appears. This enables the user to provide a Service Name.

Figure 2 The Service Name Page



If the `config.xml` contains a `<connection-factory>` entry, the Custom Adapter Wizard displays a Connection Information page that displays the default Connection Factory Location.

Figure 3 Adapter Configuration Wizard Connection Information Page

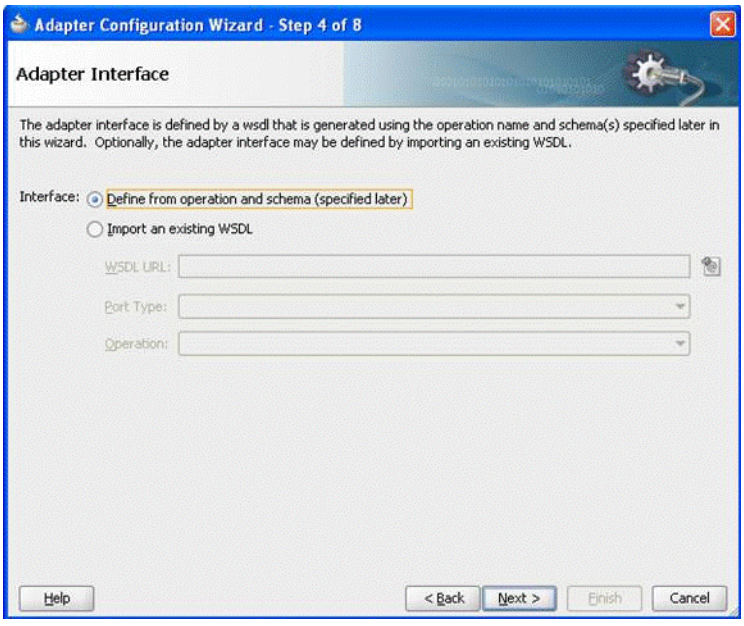


The Adapter does not require a `<connection-factory>` entry. If the `config.xml` file does not contain a `<connection-factory>` entry, this page does not appear when the user runs the Adapter Wizard.

The Custom Adapter Interface screen enables the user to either provide the name of an operation and schema to generate a WSDL, or to import an existing WSDL

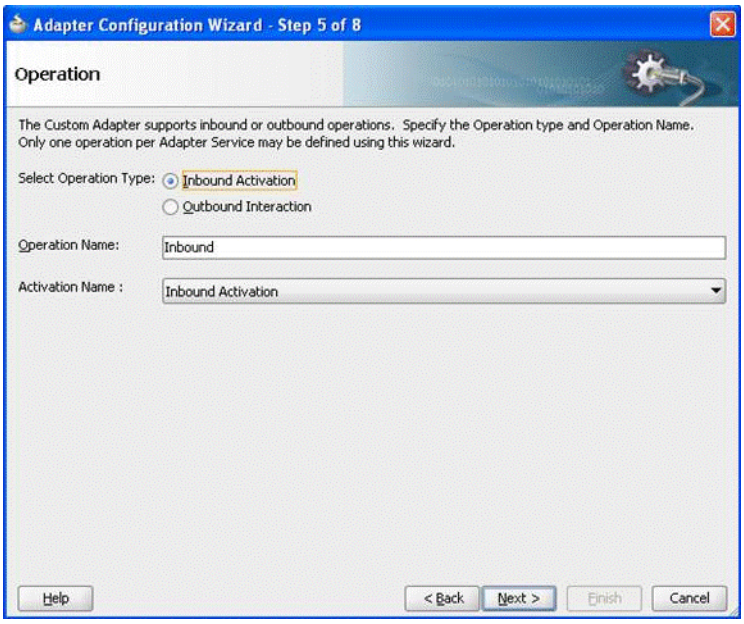
If the user chooses the latter, the URL of a WSDL, Port Type and Operation must be provided as in the screenshot below.

Figure 4 The Adapter Configuration Wizard Adapter Interface Screen



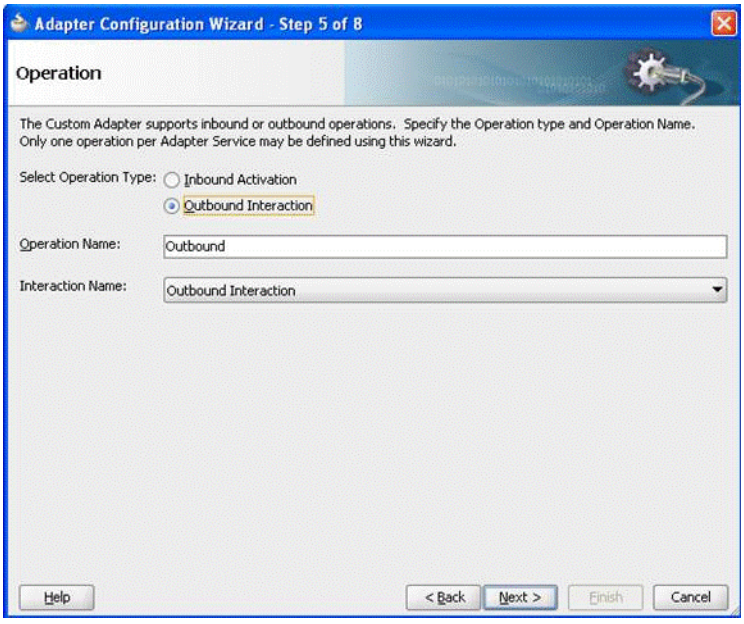
If the user selected Inbound Activation on the Custom Adapter's Operation page, the user is provided a list of Activation Class names from the `customAdapter-config.xml` file (or translated display names as seen in this example) from which to choose.

Figure 5 Adapter Configuration Wizard Displaying Operation Name and Activation Translated Display Name for an Inbound Operation



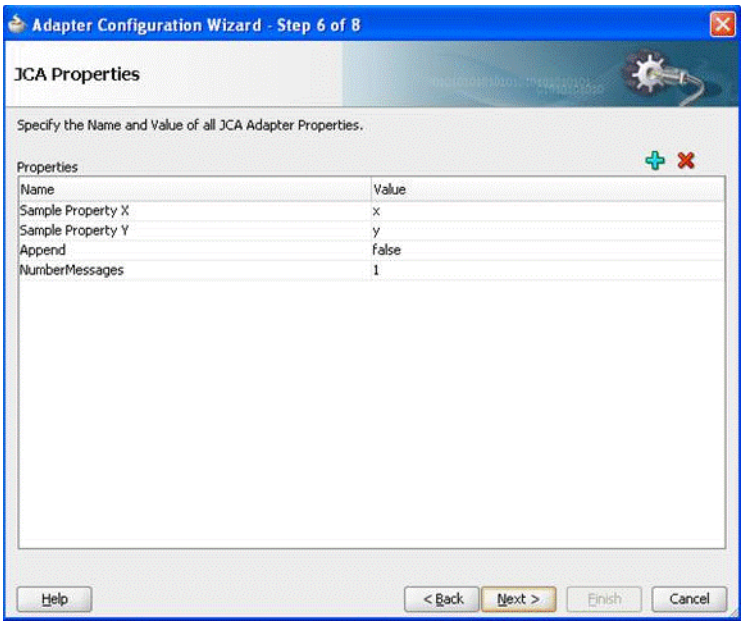
If the user selects Outbound Interaction on the Custom Adapter's Operation page, a list of Interaction Class names (or translated display names as seen in this example) is displayed from the `customAdapter-config.xml`, from which the user can choose.

Figure 6 Adapter Configuration Wizard Operation Screen Displaying Outbound Interaction with Operation Name and Interaction Name



The next page shows properties associated with that class in the generated `customAdapter-config.xml`, depending on the Class Name selected in the previous dialog.

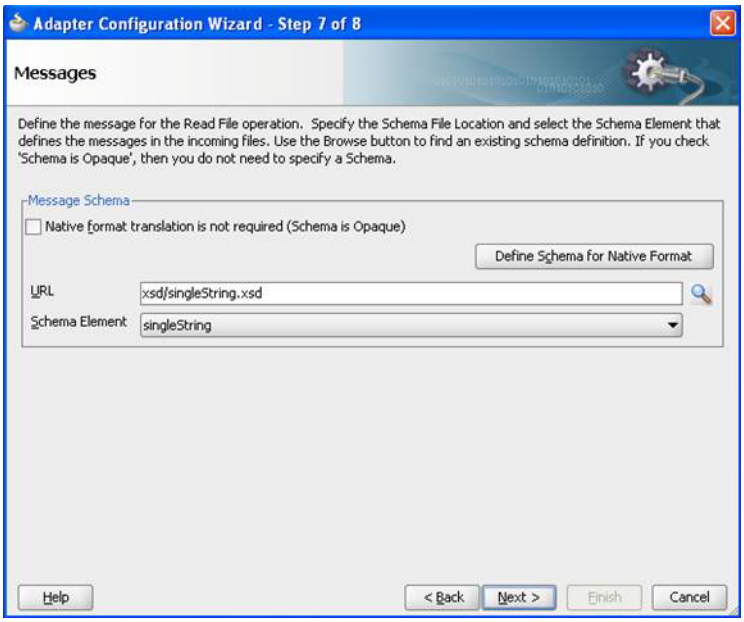
Figure 7 Adapter Configuration Wizard JCA Properties Screen



The user can change any default values and add or delete properties on this page.

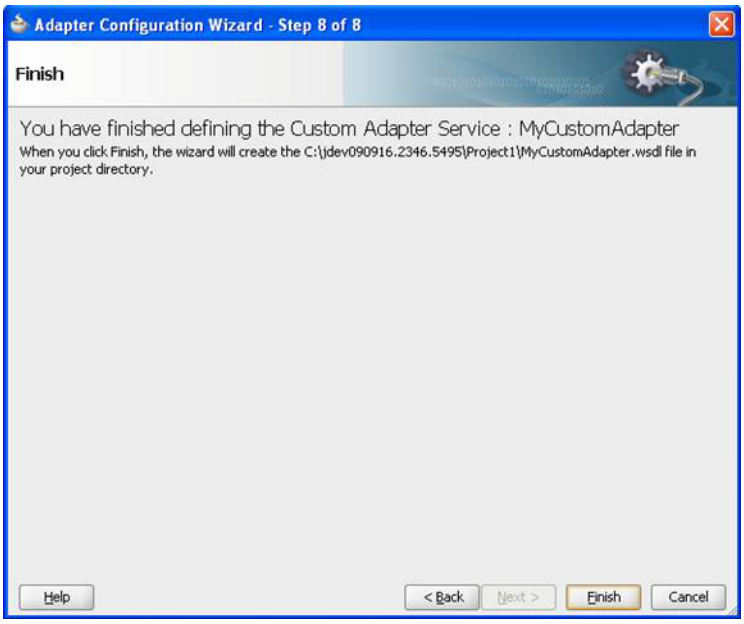
The Messages page has the same functionality in a Custom Adapter as it would work in other adapters that have the Messages page. It includes a checkbox to indicate if Native format is required, a box to define the schema for native format, the URL for the Message Schema and the Schema Element.

Figure 8 The Custom Adapter Wizard Messages Page



When the user selects Finish on the Messages Page, the new Custom JCA Adapter appears in the SOA diagram. in the JDeveloper window, and the Finish page displays.

Figure 9 The Custom Adapter Wizard Finish Page



If the user double-clicks on the Custom JCA Adapter in the SOA diagram, he/she reenters the same Adapter wizard. This enables the user to modify values that were previously entered.

3 Developing a Custom Adapter Using the Oracle JDeveloper

You can develop your own Custom Adapter using the Oracle JDeveloper. To do so, you need to understand the basic development Framework.

3.1 Understanding the JCA Adapter Wizard Framework and Overview

The `JcaAdapterWizard` extension allows you to add the initial wizard pages. You can add additional wizard pages at any later time. For example, most adapter wizards add their pages after the operation type (inbound/outbound) is selected. If the user changes his mind and selects a different operation type, the wizard pages are removed and new ones added.

Each wizard page collects user input. In addition, an adapter-specific context can be registered to enable sharing information between wizard pages.

When the user selects the Finish button, the framework invokes the `buildDataModel()` method on each wizard page in the same order that the pages were displayed. The `buildDataModel()` on each page is responsible for copying its screen data to the data model.

The Framework creates artifacts from the data model. Adapters can also register an interface to be invoked before or after the framework's finish processing.

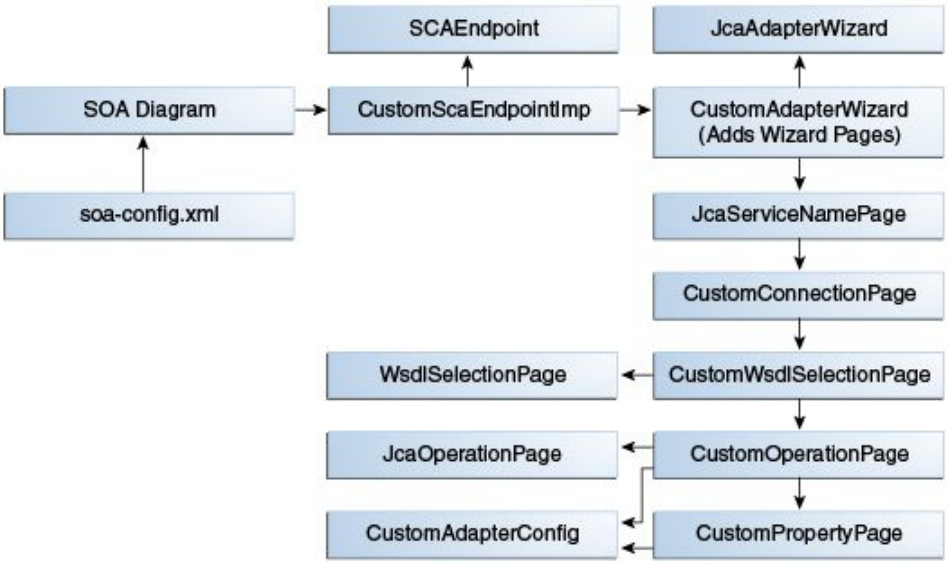
Note: The Adapter framework handles all WSDL generation. The (old) `buildWsdl()` method is still invoked for each wizard page, but most wizard pages should not override this method and leave the WSDL processing to the super class.

The following table describes, and the diagram shows the different classes that makes up the Custom Adapter. A Class Diagram follows.

Table 1 Custom Adapter Classes

Class	Description
SOA Diagram	The graphical editor for the composite.xml.
SCA Endpoint	Provides a proxy between the SOA diagram and the Adapter framework.
JcaAdapterWizard	Allows you to add initial Wizard pages.
CustomAdapterWizard	Extends JcaAdapterWizard to enable overriding public methods
JcaServiceName Page	Provides the Service Name page.
CustomConnectionPage	Creates a page that will prompt for the connection-factory location and puts this information into the data model.
CustomWSDLSelectionPage	Provides the user the option to select a existing WSDL, portType, and operation rather than generating a default WSDL
CustomOperationPage	Add and remove wizard pages depending on the operation chosen.
CustomPropertyPage	Enables the user to set activation and interaction spec properties.

Figure 10 Custom Adapter Class Diagram



Once you understand the basics, you can proceed to understand the implementation details:

- Implementing the SCA Endpoint Interface
- Extending the JCA Adapter Wizard Class and Implementing its Abstract Methods.
- Implementing Custom Adapter Wizard Pages
- Implementing the JcaDataInterface
- Using Framework Pages for Common Functionality
- Extending the ConnectionPage
- Extending the WsdlSelectionPage
- Extending the JcaOperationPage
- Using and/or Extending CustomAdapterConfig Class
- Extending CommonAdapterSchemaPage or CommonAdapterInOutSchemaPage
- Adding a Finish Page
- Performing Post Finish Handling (Optional)
- Adding an AdapterType Element to sca-config.xml
- Using Public Utility Methods

3.1.1 Implementing the SCA Endpoint Interface

The `SCAEndpoint` is invoked when a new adapter is dropped onto the SOA diagram from the JDeveloper Component Palette (`SCAEndpoint.createImplementation`), when an existing adapter is doubled-clicked within the SOA Diagram within JDeveloper (`setInterfaceInfo` and `displayServiceEditor`), or when an adapter is deleted from the SOA Diagram (`SCAEndpoint.delete`).

The `SCAEndpoint` implementation is the proxy between the SOA diagram and the Adapter framework. It is responsible for invoking the main constructor for the adapter wizard.

The `createEndpoint` method and `updateEndpoint` method thus invoke the `display()` method on the `JcaAdapterWizard` extension class.

See `oracle.tip.tools.ide.pm.wizard.adapter.custom.CustomScaEndpointImpl.java` for an example of the above.

3.1.2 Extending the JCA Adapter Wizard Class and Implementing its Abstract Methods.

You enable Adapter wizards to use the JCA Adapter Framework by extending this `oracle.tip.tools.ide.pm.wizard.adapter.jca.JcaAdapterWizard` class and by implementing its abstract methods.

Abstract methods in this class include `getAdapterTypeString()` and `getWelcomeMessage`, which fill in text needed for the generic wizard pages.

The abstract method `addInitialWizardPages()` adds Adapter wizard pages. This method should only add initial pages; that is, pages that are always navigated to regardless of user input.

The initial pages usually start with the JcaServiceNamePage, a ConnectionPage (if a connection is needed), and a WsdlSelectionPage (if supported by the adapter; see detail below).

If your adapter does not support the WsdlSelectionPage, the last initial wizard page will probably be an OperationPage.

User input on the WsdlSelectionPage and the OperationPage can modify the navigation in the wizard (that is, it can modify which pages are displayed) and additional wizard pages are added when the user chooses the NEXT button on those pages.

You can optionally override public methods in JcaAdapterWizard.These include:

- Override getAdapterSpecificContext to register an adapter-specific method that can be available on all pages.
- You can register an AdapterWizardFinishInterface to be invoked when the user selects the Finish button. You register the interface by overriding the getAdapterWizardFinish method in your JcaAdapterWizard extension class.
- You can override readJCADataModel to access your own data model rather than using the default data model. However, this is not recommended.

See oracle.tip.tools.ide.pm.wizard.adapter.custom.CustomAdapterWizard.java in <JDEV_HOME>\jdeveloper\integration\adapters\samples\custom for an example of overriding public methods.

3.1.3 Implementing Custom Adapter Wizard Pages

All adapter wizard pages must extend techAdapterWizardPage (which inherits from Jpanel).

The page constructors must invoke setContextAndPage() which adds the page to the Wizard and sets up other context information that is needed. See the example in oracle.tip.tools.ide.pm.wizard.adapter.custom.CustomConnectionPage.java

When the user chooses the Finish button, the Adapter Framework invokes the buildDataModel() method on each wizard page in the same order that the pages were displayed. The buildDataModel() method is responsible for copying its screen data to the data model. You can do this simply with code similar to the following:

```
JcaDataInterface jdata = (JcaDataInterface)getDataModel();
jdata.setProperty(propname,screenField.getText().trim());
```

The JcaAdapterContext is available on all pages. It has many accessor methods for obtaining useful information. Following is a partial list of accessors that will be commonly used by wizard pages:

Table 2 JcaAdapterContext Accessor Methods

Accessor Method	Notes
`Object getAdapterSpecificContext();`	The Adapter context object.
`JcaDataInterface getDataModel();`	Obtains the data model
`Project getProject();`	Obtains the jdeveloper Project object that is needed for most jdev interactions. It contains the project name also.

Table 2 (Cont.) JcaAdapterContext Accessor Methods

Accessor Method	Notes
`Frame getAdapterFrame();`	This returns a Frame object that is required when invoking new dialogs or pop-up error messages.
`Frame getAdapterFrame();`	This returns a Frame object that is required when invoking new dialogs or pop-up error messages.
`boolean isUpdateMode();`	Indicates that an existing Adapter is being updated
`String getServiceName();`	Obtains a service name
`String getServiceType();`	Obtains a service type
`WsdlInfo getExistingWsdlInfo();`	Null if the existing WSDL is not selected. This would be the WsdlInfo for the Wsdl/porttype the user selected on the WsdlSelectionPage.
`String getOperationName();`	Obtains an Operation name
`Boolean isInbound();`	True if the adapter is inbound (Read, Get)
`Connection getConnection();`	Obtains a connection for adapters that use a connection object, like the Database adapter.

See the `oracle.tip.tools.ide.pm.wizard.adapter.custom.CustomPropertiesPage.java` for an example of a wizard page developed from scratch.

3.1.4 Implementing the JcaDataInterface

Most of the methods of the `JcaDataInterface` have two versions:

- One version that is "spec aware" and which takes an (activation or interaction) "spec" argument that indicates the activation or interaction spec to which a property or attribute belongs.
- Another version is "non-spec aware" and does not have a spec argument. All the examples in this document use the "non-spec aware" APIs.

The "non-spec" versions of these methods are for adapter wizards that only support a single spec at a time. For example, a FileAdapter instance can either Read a file or Write to a file. A single instance cannot do both.

The "spec aware" methods must be used by adapter wizards that support multiple specs. For example, the MQ adapter supports a read operation that also has an asynchronous callback.

In this case, the MQ adapter wizard generates an activation spec *and* an interaction spec. Therefore, the MQ adapter wizard must use the "spec aware" methods to make it clear *which* spec a property or attribute belongs to.

The spec argument contains only the spec String.

The "non-spec" API also supports older adapters where the properties can be set before you add the activation or interaction spec.

3.1.5 Using Framework Pages for Common Functionality

The Welcome Page, the Service Name Page, and the Finish Page are common to all adapter wizards. These pages are controlled by the Adapter framework, so you do not

need to add them to the wizard, but there are accessor methods for overriding the contents of the Welcome Page (in the `JcaAdapterWizard`) and the Finish Page.

Framework pages must be extended by the adapter when you use them. A discussion of each type of extension follows.

3.1.6 Extending the ConnectionPage

Most, but not all, adapter wizards require a connection page. For example, the AQ adapter requires a database connection page to enable the user to query and display a list of available queues.

If your adapter wizard requires a database connection, add `JcaDBConnectionPage` in `addInitialWizardPages` and use the method as is. The JDeveloper connection objects will be available to subsequent pages by the following accessor in the `AdapterWizardContext`:

```
Connection getConnection();
DatabaseConnectionInfo getDBConnectionInfo();
```

The Custom Adapter in this guide is an example of an adapter that has its own connection type and does not use the `JcaDBConnectionPage`.

In this case, the `CustomConnectionPage` prompts for the connection-factory location and puts this information into the data model with code similar to:

```
JcaDataInterface jdata = (JcaDataInterface)getDataModel();
    jdata.setConnectionFactoryAtrribute(JcaDataInterface.
     CONNECTION_FACTORY
      _LOCATION_ATTRIBUTE,connectionTextField.getText().trim());
```

3.1.7 Extending the WsdlSelectionPage

Many adapter wizards (such as wizards for the File, FTP, AQ, MQ, and JMS adapters) provide the user the option to select a existing WSDL, portType, and operation rather than generating a default WSDL Picking an operation essentially defines the schema that will be used by the adapter.

However, selecting an existing WSDL is not supported by adapters that supply or generate their own schemas (for example, the DB, B2B, BAM, and Oracle Application adapters generate their own schemas).

Some adapter wizards are a mix of the two: if an existing WSDL is selected, then certain operations that have required schemas are disabled.

If your adapter needs to support existing WSDLs, you need to create a class that extends `WsdlSelectionPage`.

See `CustomWsdlSelectionPage` for an example. There are three abstract methods that must be implemented to extend this class:

- `boolean adapterSupportsSynchRead()`. Returns true if the adapter has a sync read option. This enables WSDL operations with an output element to be selected on the WsdlSelection page.
- `boolean adapterSupportsCallback()`. Returns true if the adapter can have a callback (inbound or outbound). If true, the `WsdlSelection` page enables the user to choose two port types.
- `createPagesForOperation()`. Enables you to add new wizard pages after the user chooses the WSDL operation.

Note: Subsequent wizard pages must disable operations and features that are not supported by the selected operation. For example, with the File and FTP adapters, if the operation selected has an output element, all operations on the Operation page must be disabled except the Synchronous Read operation. See additional details under `JcaOperationPage`.

When an existing WSDL is selected, the adapter framework still generates a WSDL, but rather than generating port types, operations, messages, and schemas in the WSDL, the framework imports the WSDL the user has selected. A wrapper WSDL that will import the selected WSDL. It only contains the import and the partnerlinkTypes.

Generating a wrapper WSDL is similar in approach to that of BPEL when the user selects a WSDL that does not have a `partnerLinkType`.

The adapter wrapper WSDL also defines `partnerLinkTypes` and can define a header message that is not available in the user-selected WSDL

3.1.8 Extending the JcaOperationPage

All wizard operation pages (pages that pick operations such as Read/Write or Get/Put) should override `JcaOperationPage` because `JcaOperationPage` handles the WSDL object creation, based on the operation name.

The operation name is provided by implementing the abstract method `getOperationName()`.

Most operation pages add and remove wizard pages depending on the operation chosen. The accessor method `removePages()` removes all wizard pages past the current page.

The Operation page's `buildDataModel()` method is responsible for setting the Activation or Interaction spec in the data model, using code similar to:

```
JcaDataInterface jdata = (JcaDataInterface)getDataModel();
    if (readRB.isSelected()){
        jdata.setActivationSpec
          (THIS_ADAPTERS_ACTIVATION_SPEC);
    )
    else{
        jdata.setInteractionSpec(THIS_ADAPTERS_INTERACTION_SPEC);
    }
```

Note: For update mode, `OperationPages` must not enable the user to change operation types or names because the adapter interface can be wired to a reference that depends on the existence of that operation.

See `oracle.tip.tools.ide.pm.wizard.adapter.custom.CustomOperationPage.java` for an example.

3.1.9 Using and/or Extending CustomAdapterConfig Class

Some adapters can use or extend the `CustomPropertyPage` to enable the user to set their activation and interaction spec properties.

But most new adapters have their own property pages because this provides the Adapters with more control of how the properties display (for example, a list of values, a UI control, or a validation).

The class that the Custom adapter uses to read its configuration file, `CustomAdapterConfig`, is public and can be used or extended by other adapters.

The `CustomAdapterConfig` class retrieves lists of possible specs, properties, and connection-factories from the configuration file, `customAdapter-config.xml` (which is the default custom Adapter configuration file).

You can use a different configuration file by extending this class and overriding the `getConfigFilePath()`method.

The attributes `displayResourceKey` and `resourceBundle` are optional.

If activation-spec, interaction-spec, or property elements have a `displayResourceKey`, The framework uses the attribute value as a key to retrieve displayable text from a resource bundle available by the `getDisplayString()` or `toString()` methods.

If a resource bundle is not available or the key is not found in the bundle, the key itself is used as the displayable text (hence, the class is not required to have a resource bundle).

You can pass a resource bundle to the constructor of this class, but it also can be overriden by the `resourceBundle` attribute on the <connection-factory> element in the configuration file.

3.1.10 Extending CommonAdapterSchemaPage or CommonAdapterInOutSchemaPage

Many adapter wizards enable the user to chose a single schema or request/reply schema to define the messages used by the adapter.

These wizards extend the `CommonAdapterSchemaPage` (for adapters with one-way operations) or `CommonAdapterInOutSchemaPage` (for adapters with both one-way and request/reply operations), with options related to Schema (typeChooser, NXSD wizard, opaque schema, and WSDL generation).

The `CommonAdapterSchemaPage` class extension only needs to implement `getIntroPrompt()` to change the introductory text at the top of the page.

See `oracle.tip.tools.ide.pm.wizard.adapter.custom.CustomSchemaPage` for an example.

The `CommonAdapterInOutSchemaPage` has several public methods that you can override. See the following table.

Table 3 CommonAdapterInOutSchemaPage Public Methods

Method	Description
`getIntroPrompt()`	Changes the introductory text at the top of the page.
`getInboundSchemaLabel()`	Sets the label for the Inbound Schema Panel.
`getOutboundSchemaLabel()`	Sets the label for the Outbound Schema Panel.
`changeSequenceOfInOutSchemaPanels()`	Changes the sequence of display of Inbound and Outbound panels. Default display is Inbound schema panel first.
`setInOutSchemaPanelVisibility()`	Sets or unsets schema panel visibility.

See `oracle.tip.tools.ide.pm.wizard.adapter.mq.MQAdapterInOutSchemaPage` for an example.

3.1.11 Adding a Finish Page

You must add a Finish page to the wizard early in the wizard flow, rather than having the wizard display the Finish button when the Finish page is displayed; the Finish page displays summary information needed from other pages that have not yet been displayed.

To help provide this order of page precedence, there are two helper methods contained in `techAdapterWizardPage`: `addFinishPage()` and `setFinishPageContents()`:

- `addFinishPage()` adds a finish page with no contents to the wizard. Most wizards add the Finish page when they remove and add pages
- `setFinishPageContents(String title, Object contents)` calls `setFinishPageContents` when the user chooses the Next button on the last page of the wizard (from `wizardValidatePage()`)

The contents of the Finish page can be a Component or a String. If you set the contents of the Finish page to be a String, a MultiLineLabel is created for you.

3.1.12 Performing Post Finish Handling (Optional)

Each Adapter can register an implementation of the `AdapterWizardFinishInterface` by overriding the `getAdapterWizardFinish()` method in the `JcaAdapterWizard` extension class. This enables the adapter to perform additional activities at Finish time.

For example, the File adapter uses this Finish override to check if a `FileAdapterHeader` schema is in the current project and, if the schema is not in the current project, the File Adapter copies the Header file to the project directory.

The `AdapterWizardFinishInterface` has two methods that either obtain control before or after the adapter artifacts (WSDL and `.jca` files) are created:

- The `beforeWSDL()`method returns false to indicate that the adapter artifacts should not be created.
- The `afterWSDL`() method is invoked the after WSDL file is created.

Both of these methods can be used as you require in your implementation.

3.1.13 Adding an AdapterType Element to sca-config.xml

The following, added to the `sca-config.xml`, tells the SOA diagram the icons and the SCAEndpoint implementation class to use for the Custom Adapter, and provides additional information. Sub-element definitions include:

- name: from the resource bundle
- description: from the resource bundle
- tooltip: from the resource bundle (shows under mouse cursor over the graphical shape)
- icon16x16: the icon used in the palette
- icon20x20: the icon used in the graphical shape
- remaining 4 icons are not used

```
<adapterType
     resourceBundle="oracle.tip.tools.ide.pm.modules.biz
```

```
        integration.adapter.custom.resource.CustomStringResourceBundle">
          <name>${CUSTOM_ADAPTER_COMPONENT_NAME_L}</name>
          <bindingType>jca</bindingType>
           <bindingSubType>custom</bindingSubType

<implementationClass>oracle.tip.tools.ide.pm.modules.bizintegration.adapter.custom
.CustomScaEndpointImpl</implementationClass>
      <description>${CUSTOM_ADAPTER_COMPONENT_DESC}</description>
      <tooltip>${CUSTOM_ADAPTER_COMPONENT_DESC}</tooltip>

<icon16x16>/oracle/tip/tools/ide/pm/modules/bizintegration/adapter/custom/resource
/custom_adapter_16x16.png</icon16x16>

<icon20x20>/oracle/tip/tools/ide/pm/modules/bizintegration/adapter/custom/resource
/custom_adapter_20x20.png</icon20x20>
      <topSectionIcon>oracle/tip/tools/ide/fabric/resource/image/visuals_
rd1/whiteServiceTop.png</topSectionIcon>
      <middleSectionIcon>oracle/tip/tools/ide/fabric/resource/image/visuals_
rd1/whiteServiceMiddle.png</middleSectionIcon>
      <bottomSectionIcon>oracle/tip/tools/ide/fabric/resource/image/visuals_
rd1/whiteServiceBottom.png</bottomSectionIcon>
      <collapsedSectionIcon>oracle/tip/tools/ide/fabric/
      resource/image/visuals_rd1/whiteServiceCollapsed.png</collapsedSectionIcon>
    </adapterType>
<collapsedSectionIcon>oracle/tip/tools/ide/fabric/resource/image/
visuals_rd1/whiteServiceCollapsed.png</collapsedSectionIcon>
  </adapterType>
```

In the `sca-config.xml`, the <adapterType> has a resource bundle attribute that must contain the class name of the bundle from which the SOA diagrammer uses to get the translated text it uses to label endpoints in the diagram.

For example, referring to the Custom Adapter entry, the diagram will look up `CUSTOM_ADAPTER_COMPONENT_NAME_L` in the `CustomStringResourceBundle` when it shows the custom adapter in the diagram.

Note that internal to an adapter wizard, the diagram uses a resource bundle to get the text for all the labels in the wizard. In the cases of the custom adapter, the same bundle is being used.

3.1.14 Using Public Utility Methods

Classes in this SDK have utility methods that have been used throughout the adapter wizards. The following table lists these methods:

Method	Notes
`oracle.tip.tools.ide.adapters.designtime.adapter.jca.JcaUtil:`	
`getAdapterWsdlInfo (AdapterWizardContext_wcontext)`	Multiple flavors are available
`initContextFromWsdlAndModel(Project project,JcaAdapterContext jcaAdapterContext,WsdlInfo wsdlInfo, JcaDataInterface jdata)`	

Method	Notes
`initContextFromWSDLMultiOper( JcaAdapterContext jcaAdapterContext, JcaDataInterface jdata)`	
`Definition readAdapterWsdl(WSDLFactory wfact, URL wsdlLocation)`	
`oracle.tip.tools.ide.adapters.designtime .adapter.CommonAdapterSchemaPage: buildOpaqueSchema(AdapterWizardContext _ wcontext, Definition def1)removeOldSchema( AdapterWizardContext wcontext, String schemaLoc,String schemaNamespace)`	Removes schema from definition object for a one-way operation
`removeOldSchema(AdapterWizardContext wcontext, boolean oneway, boolean isCallback)`	Removes input/output schema from definition object for request/reply operation

4 Developing an Adapter Runtime Component for Integration with Fusion Middleware

There are several design and connection configuration steps involved with developing an adapter runtime component.

These include:

- Understanding Overall Design
- Connection Configuration
- Run Time Interfaces and Contracts
- Constructing the WebLogic JCA Resource Archive RAR
- Deploying the RAR File to the WebLogic Application Server
- Testing the Custom Adapter

4.1 Understanding Overall Design

The Oracle JCA Framework provides pluggability for adapters that comply to the J2EE Java Connector Architecture, version 1.5. That is, any adapter used with the Oracle JCA Framework must fulfill the requirements documented in the JCA 1.5 specification (see also JSR-112, J2EE Connector Architecture 1.5 - Final Release).

> **Note:** The Oracle JCA Framework implements a lightweight JCA 1.5 container, which enables dynamic (code-driven) deployment of inbound endpoints (activations). This does mean a small disadvantage in precluding support for JCA 1.5 Transaction Inflow (that is, support for XATerminator support).

4.2 Connection Configuration

The Oracle JCA Framework requires a JCA adapter that is deployable on a J2EE 1.4- (or later) compliant Application Server, for example, Oracle WebLogic Server. Hence, any JCA adapter used with the Oracle JCA Framework must implement all the

relevant JCA SPI (Service Provider) and CCI (Common Client) interfaces as mandated by the JCA 1.5, specification, principally the following:

- `javax.resource.spi.ManagedConnectionFactory`
- `javax.resource.cci.ConnectionFactory`
- `javax.resource.spi.ManagedConnection`
- `javax.resource.cci.Connection`

The JCA adapter must define the necessary connection properties in its `ManagedConnectionFactory` to establish a functional connection to an EIS (Enterprise Information System, or "back-end").

Properties in the `ManagedConnectionFactory` enable the Oracle JCA Framework to obtain a connection via a deployed JCA Adapter. Example code is:

```
ConnectionFactory cf = initialContext.lookup("eis/Siebel/SiebelApp1-Connection");
Connection c = cf.getConnection();
```

All connection properties are retrieved through the JNDI lookup contained in the `ConnectionFactory` instance.

4.3 Run Time Interfaces and Contracts

The Adapter must comply with specific interfaces and contracts to successfully interact with the Oracle JCA Framework. Principally, the JCA adapter must implement at least the following interfaces:

- `javax.resource.spi.ResourceAdapter`
- `javax.resource.spi.ActivationSpec`
- `javax.resource.spi.work.Work`
- `javax.resource.cci.Interaction`
- `javax.resource.cci.InteractionSpec`

These SPI interfaces must be implemented to enable inbound message flow, and the CCI interfaces must be implemented to enable outbound interactions. See the following Section 4.3.1, "Inbound" for information about Inbound interfaces.

4.3.1 Inbound

The J2EE Java Connector Architecture, version 1.5, provides interfaces that enable an adapter to asynchronously push messages to endpoints managed by the J2EE Application Server in a well-defined manner in terms of transactions (that is, when an adapter supports transactional semantics).

In the Oracle JCA Framework, these endpoints are managed by the Adapter Framework.

4.3.2 JCA 1.5 Contracts and Interfaces

The following sub-sections describe each of the Service Provider Interfaces (SPI) that a JCA 1.5 compliant Resource Adapter must implement in order to be deployable to Oracle Fusion Middleware.

4.3.2.1 Interfacejavax.resource.spi.ResourceAdapter

Each JCA adapter must implement this interface. The Adapter Framework treats an Adapter's implementation of this interface as a singleton; that is, there will be *at most* be one instance of an implementation of this interface per Java Virtual Machine.

If an adapter is not used by *any* business process (in particular JVM) then it's ResourceAdapter implementation will not be instantiated

4.3.2.2 Instantiation

The ResourceAdapter implementation is instantiated when the first composite application using the adapter is started.

Subsequent starting composites that also use the same adapter do not cause additional `ResourceAdapter` instances to be created, but obtain a handle to the first (singleton) instance.

The following method is called immediately after the `ResourceAdapter` instance is created:

```
public void start(BootstrapContext ctx)
   throws ResourceAdapterInternalException
```

The adapter should cache the `BootstrapContext`, as it contains necessary facilities for creating and scheduling inbound endpoint interactions.

The `BootstrapContext` also contains a Logging service handle, which the resource adapter should cache and use throughout all classes supporting inbound message flow (that is. classes provided under `javax.resource.spi.*`).

Outbound, the Logging service handle will be available through the ConnectionFactory.

4.3.2.3 Stop Method and Endpoint Activation

The `stop` method is called when a resource instance is undeployed, during application server shutdown, or when the last composite referring to the adapter is terminated.

```
public void stop()
```

If active endpoints exist, they must be deactivated before returning from the `stop()` method, invoked by the adapter framework during server or composite shutdown.

Any other allocated resources must similarly be released promptly:

```
public void endpointActivation(MessageEndpointFactory endpointFactory,
       ActivationSpec spec)
```

Endpoint activation is called during the activation of a message endpoint.

Each composite that has an initiating or non-initiating JCA-based Service entry point (that is, an entry point using `binding.jca` in the `<service>`) causes the invocation of `endpointActivation()` for each associated (inbound) operation for the ResourceAdapter referenced by the connection factory JNDI in the `.jca` property file.

4.3.2.4 Connection Factory, Work Request, Endpoint Deactivation

The Adapter Framework provides, through the `MessageEndpointFactory`, an instance of the `ConnectionFactory`, as defined in the `<connection-factory>` element in the `.jca` property file, which then can be used by the resource adapter to create connections:

```
javax.resource.cci.ConnectionFactory connectionFactory =
```

```
    ((ConnectionFactoryAssociation)activationSpec).getConnectionFactory();
javax.resource.cci.Connection connection = connectionFactory.getConnection();
```

During this invocation, the resource adapter must use the `MessageEndpointFactory` to create the inbound endpoint and to submit a Work request for execution, which constitutes the inbound thread that monitors the inbound EIS endpoint.

```
ResourceAdapterInboundWorkerThread workRequest =
    new ResourceAdapterInboundWorkerThread(endpoint, activationSpec, connection);
      workManager.startWork(workRequest);
```

Here `ResourceAdapterInboundWorkerThread` is the resource adapter's implementation of `javax.resource.spi.work.Work`.

Soon after returning from `startWork()`, the Adapter Framework allocates and assigns a thread to the submitted `workRequest` by calling its `run()` method.

The following method, endpointDeactivation, is called by the Adapter Framework when a message endpoint is deactivated; that is, either when the composite having activated the endpoint shuts down, or when the application server shuts down.

```
public void endpointDeactivation(MessageEndpointFactory
  endpointFactory, ActivationSpec spec)
```

4.3.3 Interfacejavax.resource.spi.work.Work

The Work interface is:

```
public interface Work extends Runnable
```

Because a JCA 1.5-compliant resource adapter deployed with the Oracle SOA will be executing in Managed Mode (that is, inside a J2EE container), it cannot create threads on its own. Rather, the adapter must rely on the WebLogic Application Server to create and start threads on its behalf.

To obtain a thread (for example, a thread to be used to poll an inbound endpoint), the adapter must submit an instance of a class implementing the Work interface to the WorkManager (which in turn is obtained via the `BootstrapContext`).

The adapter does this by the `run` method:

```
public void run()
```

This method is invoked by the WorkManager, using a newly allocated J2EE-compliant thread. The resource adapter can use this thread until it chooses to stop (for example, due to a unrecoverable error condition; however, this convention is not recommended) or, more appropriately, until the adapter is signalled to stop (by the `release()` method):

```
public void release()
```

The adapter itself invokes `release()` when it is processing the invocation of `endpointDeactivation` made by the Adapter Framework.

This activity is called on a separate thread from the one currently executing the Work instance, that is, from the system thread invoking `endpointActivation`.

If the resource adapter does not exit the `run()` method after a preset time following the invocation of `release()`, the Adapter Framework attempts to forcefully stop the thread.

4.3.4 Interfacejavax.resource.spi.work.WorkManager

In Oracle SOA, the implementation of the `WorkManager` interface is provided by the Adapter Framework. The implementation is minimal, because it does not support advanced thread pooling or sophisticated scheduling.

Instead, the Adapter Framework implements only one of six public methods, `scheduleWork(Work work)`. The other public methods are redirected to this method, that is, call blocking is not supported (as for example, it is required by `doWork()`).

The Adapter Framework leverages the SOA default Work Manager, enabling threads freed from a finished `Work` instance to be reused in new Work submissions.

The following method accepts a Work instance for processing.

```
public void scheduleWork(Work work)
   throws WorkException
```

This call does not block and returns immediately once a Work instance has been accepted for processing. There is no guarantee when the submitted Work instance starts execution; that is, there is no time constraint to start execution.

4.3.5 Outbound Considerations

In outbound and inbound scenarios, the Resource Adapter must comply to the standard contracts defined by the JCA 1.0 specification.

In the outbound scenario, when providing the runtime definition of an adapter, you must adhere to the following requirements:

- The `ManagedConnectionFactory` of the Resource Adapter must define all of its connection related parameters, thereby allowing the Oracle SOA runtime to simply perform. For example:

  ```
  ConnectionFactory cciConnectionFactory =
     initialContext.lookup("eis/sample/...");
  Connection cciConnection = cciConnectionFactory.getConnection();
  ```

- All regular `InteractionSpec` parameters on any `InteractionSpec` implementation must be constituted by single argument mutator methods, using one of the following argument types: String, int, boolean, short, long, float, double.
- The Resource Adapter must be able to function in a Managed Environment, and as such it must be deployable on the J2EE Application Server.

4.3.6 Translation Service Interface

The Translation Service interface handles translation of a message to and from native formats. The Adapter Framework determines any operation-related message Translation to- or from- native format requirements by inspecting the `InteractionSpec` or `ActivationSpec` for a given WSDL Binding Operation.

If one of the indicated Specs implement the Translation marker interface `TranslationAware`, the Adapter Framework supplies the corresponding XSDElement from the WSDL types section, which defines the input message type.

Specifically, the Adapter Framework performs the following steps:

```
jcaInteractionSpec=(InteractionSpec)
  Class.forName(_interactionSpecName).newInstance();
    if (jcaInteractionSpec instanceof TranslationAware)
      {
            oracle.xml.parser.schema.XMLSchema nXsdSchemaRoot =
```

```
                getInputMessageSchemaElement();
                ((TranslationAware)jcaInteractionSpec).
                   setNXSDSchemaRoot(nXsdSchemaRoot);
                   }
jcaInteraction.execute(jcaInteractionSpec, ...
```

4.4 Constructing the WebLogic JCA Resource Archive RAR

To deploy a JCA adapter on a WebLogic Application Server, the user must create a Deployment Descriptor file that defines the following properties:

- The display name
- The adapter vendor
- The EIS/back-end type
- The version numbers
- The class name of adapter implementation of `javax.resource.spi.ManagedConnectionFactory`
- The class name of the adapter implementation of `javax.resource.cci.ConnectionFactory`
- The class name of adapter implementation of `javax.resource.cci.Connection`
- The names of all bean properties exposed in the `ManagedConnectionFactory` implementation.

The deployment descriptor only defines an instance of `<outbound-resourceadapter>` because the inbound JCA runtime container is directly managed by the Oracle JCA Framework rather than by the Oracle WebLogic Server.

The deployment descriptor file (`ra.xml`) must be archived into a RAR file (resource archive), which then can be deployed using the WebLogic Server Console or through scripting.

The RAR file that the user deploys must have the following structure:

```
/META-INF/ra.xml
/adapter.jar
/dependencies.jar
```

The `adapter.jar` file contains the implementation of the JCA adapter, and `dependencies.jar` contains all dependent Java libraries that are not implicitly provided by the WebLogic Application Server.

Once these constituent files have been placed in the above directory structure, you can create the RAR file using the command:

```
$ jar cf myAdapter.rar META-INF/ra.xml adapter.jar dependencies.jar
```

See the appendix for a list of jar files that are required to compile a project.

4.5 Deploying the RAR File to the WebLogic Application Server

After creating the adapter resource archive, the user must deploy the adapter on the WebLogic Application Server, either through the Deployment section of the WLS Management Console, or by using the WLST tool (see

http://docs.oracle.com/cd/E13222_01/wls/docs90/config_scripting/reference.html#1024285).

The RAR file can also be directly copied to the `autodeploy` directory under the WebLogic Server domain directory, where it will be automatically deployed.

After having deployed the RAR, the user can create connection factories under the WebLogic Server Management Console Deployment section. See the following screenshot for an example, using the Outbound Connection Pool Configuration Table.

Figure 11 Creating Connection Factories using the WebLogic Server Management Console

4.6 Testing the Custom Adapter

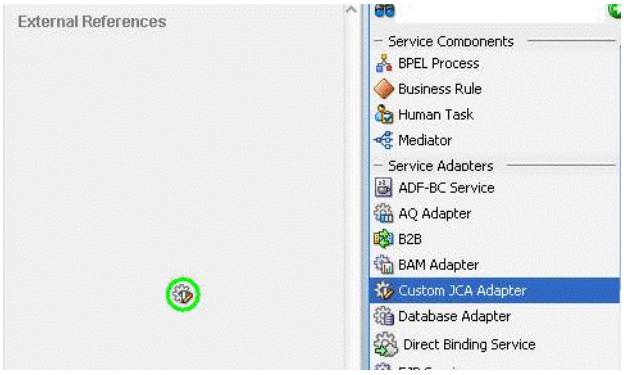
See the Samples section later in this document for information related to testing the Custom Adapter.

5 Making the Custom Adapter Available in Oracle JDeveloper

When the Oracle SOA Suite is installed, the Custom Adapter is not available by default in the current release. To make the Custom Adapter available in JDeveloper:

1. Edit the `JDEV_HOME\jdeveloper\integration\seed\soa\configuration\soa-config.xml` file.
2. Search for the word "custom" in this file.
3. Uncomment the `adapterType` element associated with the word custom in the file. The Custom Adapter then appears in the Component Palette for the SOA Diagram in JDeveloper, as in the following screenshot.

Figure 12 Custom JCA Adapter Appearing in JDEV Component Palette List



The `<JDEV_HOME>\jdeveloper\integration\seed\soa\configuration\customAdapter-config.xml` file contains detailed options, including connection-factory location, interaction-spec className, activation-spec className, and properties.

The properties provided in the file under an activation-spec are the properties for an inbound adapter.

Similarly. the properties provided in the file under an interaction-spec are the properties for an outbound adapter.

The property values are the default values shown by the Custom Adapter.

Users must modify the contents of `customAdapter-config.xml` to match the options needed by their Custom Adapter. For example, users can change all property names and their default values, add new properties, or even add multiple activation or interaction specs.

The `displayResourceKey` and `resourceBundle` attributes are optional. If an activation-spec, interaction-spec, or property element has a `displayResourceKey`, the attribute value is used as a key to retrieve displayable text from a resource bundle.

If a resource bundle is not available or the key is not found in the bundle, the key itself is used as the displayable text (as the key is not required to have a resource bundle).

However, you can use a resource bundle used by placing the `resourceBundle` attribute on the connection-factory element.

The contents of the sample `customAdapter-config.xml` file used in the screenshots below are.

```
<adapter-config
   xmlns="http://platform.integration.oracle/blocks/adapter/fw/metadata">
   <connection-factory location="eis/Custom/CustomAdapter"
      resourceBundle="oracle.tip.tools.ide.pm.modules.bizintegration.adapter.
      custom.resource.CustomStringResourceBundle"/>
<endpoint-interaction >
    <interaction-spec
       className="oracle.tip.adapter.custom.outbound.CustomInteractionSpec"
        displayResourceKey="CustomInteractionSpec" >
          <property name="PropX" value="x" displayResourceKey="SAMP_PROP_X" />
          <property name="PropY" value="y" displayResourceKey="Sample Property Y"/>
          <property name="Append" value="false"/>
          <property name="NumberMessages" value="1"/>
    </interaction-spec>
</endpoint-interaction>
```

```
 <endpoint-interaction >
    <activation-spec
      className="oracle.tip.adapter.custom.inbound.CustomActivationSpec"
        displayResourceKey="CustomActivationSpec">
           <property name="UseHeaders" value="false"/>
           <property name="PhysicalDirectory" value="x"/>
           <property name="Recursive" value="true"/>
           <property name="DeleteFile" value="true"/>
           <property name="IncludeFiles" value="x"/>
           <property name="PollingFrequency" value="60"/>
           <property name="MinimumAge" value="0"/>
   </activation-spec>
 </endpoint-activation>
</adapter-config>
```

6 Samples for Custom Adapter SDK Development

There are samples for source code, build scripts and composite applications that can help you understand Adapter SDK development.

6.1 Source Code

Custom adapter code and build scripts for a simple but functional adapter can be found at the following URL, and in the SOA installation JDeveloper directory, `<JAVA_HOME>/developer/integration/adapters/samples/custom`

http://java.net/projects/oraclesoasuite11g/downloads/download/Adapters/Custom/runtime-sdk.zip

6.2 Sample SCA Composite Application

A SCA composite application can help you to design a test scenario that provides a thorough test of your Custom Adapter's functionality.

If the Custom Adapter provides both inbound and outbound functionality, the test composite can employ the adapter within both an inbound and outbound (or "end-to-end" scenario.

If the Custom Adapter is unidirectional (for example. only outbound), the test composite can be triggered through a SOAP entry point by using the SOA test console.

http://java.net/projects/oraclesoasuite11g/downloads/download/Adapters/Custom/sca_project.zip

7 List of JAR Files Required for Build

The following is a comprehensive list of jar files required to perform a build:

- `oracle.sca.ui.adapters.jar`
- `xbean.jar`
- `oracle.ide.jar` from `adapter_xbeans.jar/jdeveloper/ide/extensions`
- `javatools-nodeps.jar` from `oracle_common\modules\oracle.javatools_11.1.1`
- `com.oracle.ws.orawsdl_1.3.0.0.jar` from `modules`
- `jewt4.jar` from `jdeveloper/jlib`

- oracle.sca.modeler.jar from jdeveloper/jdev/extensions
- share.jar from oracle_common/modules/oracle.bali.share_11.1.1
- bmp-ide-common.jar from developer/integration/lib
- javatools.jar from jdeveloper/ide/lib
- oracle.jdevimpl.wsdl.jar from jdeveloper/jdev/extensions

8 Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Oracle® SOA Suite, 11g (11.1.1.9.0)
E40792-02