webMethods Adapter Development Kit Installation and User's Guide

Version 6.5

July 2012

WEBMETHODS

This document applies to webMethods Adapter Development Kit 6.5 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: ADAPTER-WMK-IUG-65-20210326

Table of Contents

About this Guide **7**
Document Conventions 8
Online Information and Support 9
Data Protection 9

1 Overview **11**
What is the Adapter Development Kit? 12
Points of Integration 12
Development-Time Tasks and Support 13
Design-Time Tasks 19
The Run-Time Conceptual Model 19

2 Installing, Upgrading, and Uninstalling the Adapter Development Kit **21**
Overview 22
Requirements 22
Installing Adapter Development Kit 6.5 22
Upgrading to Adapter Development Kit 6.5 23
Uninstalling Adapter Development Kit 6.5 23

3 The Adapter Definition **25**
Overview 26
Setting Up Your Environment for Adapters 27
Adapter Definition Classes 30
Adapter Definition Implementation Classes 30
Creating a WmAdapter Implementation Class 31
Creating Resource Bundles 36
Deploying the Adapter 41

4 Connections **49**
Overview 50
Connection Classes 52
Adapter Connection Implementation Classes 53
Creating a WmManagedConnection Implementation Class 54
Creating a WmManagedConnectionFactory Implementation Class 56
Updating the Resource Bundle 62
Updating AdapterTypeInfo 62
Connection Class Interactions 62
Configuring and Testing Connection Nodes 66

5 Adapter Services **67**
Overview 68
Adapter Service Classes 68

The Metadata Model for Adapter Services....70
Adapter Service Template Interactions....89
Adapter Service Implementation....92
Configuring and Testing Adapter Service Nodes....106

6 Polling Notifications....107
Overview....108
Polling Notification Classes....109
The Metadata Model for Polling Notifications....110
Polling Notification Callbacks....110
The runNotification Method....111
Polling Notification Interactions....112
Polling Notification Implementation....115
Cluster Support for Polling Notifications....123

7 Listener Notifications....129
Overview....130
Listener Classes....131
Asynchronous Listener Notification Classes....133
Synchronous Listener Notification Classes....134
Listener and Listener Notification Interactions....135
Listener Implementation....138
Listener Notification Implementation....142

8 Design-Time Tasks....149
Overview....150
Package Management....150
Configuring Connection Nodes....153
Configuring Adapter Service Nodes....156
Polling Notification Nodes....157
Listener Notification Nodes....159

9 Run Time Activities....163
Overview....164
Retry and Recovery Architecture....164
Run Time Connection Allocation for Adapter Services....167

A Alternative Approaches to Metadata....175
Overview....176
Implementing Metadata Parameters Using External Classes....176
An Alternative Approach to Organizing Resource Domains....176
Using Resource Bundles with Resource Domain Values....180

B Integration Server Transaction Support....183
Overview....184
Simple Transactions....184
More Complex Transactions....185

Implicit Transaction Usage Cases....186
Explicit Transaction Usage Cases....187
Built-In Services For Explicit Transactions....191
Transaction Error Situations....194
Specifying Transaction Support in Connections....195

C Using the Services for Managing Namespace Nodes....197
Overview....198
Connection Services....198
Adapter Service Services....198
Listener Services....199
Listener Notification Services....199
Polling Notification Services....200

D Using the Sample Adapter....237
Overview....238
The Sample Server....239
Banking Functions....240
Banking Event Queries....240
Banking Alerts....241
Prerequisites for Code Compilation....241
Phase 1: Creating an Adapter Definition....242
Phase 2: Adding a Connection....244
Phase 3: Adding Adapter Services....249
Phase 4: Adding Polling Notifications....255
Phase 5: Adding Listener Notifications....263

About this Guide

- Document Conventions 8
- Online Information and Support 9
- Data Protection 9

This guide describes how to install, upgrade, and uninstall Adapter Development Kit, as well as how to configure and use it. This guide contains information for application developers who want to create adapters that interact with webMethods Integration Server.

To use this guide effectively, you should be familiar with:

- Terminology and basic operations of your operating system
- How to perform basic tasks with Integration Server and Software AG Designer

This version of the *webMethods Adapter Development Kit Installation and User's Guide* contains the most up to date information about the Adapter Development Kit. This version obsoletes, replaces, and supersedes all previous versions.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at http://documentation.softwareag.com.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at https://empower.softwareag.com/.

You can find product information on the Software AG Empower Product Support website at https://empower.softwareag.com.

To submit feature/enhancement requests, get information about product availability, and download products, go to Products.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the Knowledge Center.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at https://empower.softwareag.com/public_directory.aspx and give us a call.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at http://techcommunity.softwareag.com. You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 Overview

- What is the Adapter Development Kit? 12
- Points of Integration 12
- Development-Time Tasks and Support 13
- Design-Time Tasks 19
- The Run-Time Conceptual Model 19

What is the Adapter Development Kit?

The Adapter Development Kit (ADK) is a set of public Application Programming Interfaces (APIs) that you can extend to create custom adapters that interact with webMethods Integration Server. The ADK abstracts adapters from Integration Server, thus ensuring that ADK-created adapters will continue to run with future versions of Integration Server.

Like any Integration Server-based adapter, your adapter will link your back-end system with heterogeneous systems outside your organization via Integration Server, regardless of the technology at either end, and without requiring changes to the existing security infrastructure. An adapter that you create is secure and scalable because it is an add-on, system-level software component that you plug into an Integration Server. Adapters use the Integration Server application services to connect to back-end systems.

The ADK provides:

- An architecture for creating adapters based on JCA.

  This architecture supports the JCA Common Client Interface (CCI) and extends it to provide additional functionality. It includes a standard set of system-level contracts between an Integration Server and the back-end system to which the adapter connects. These contracts handle aspects of integration such as connections, adapter services, notifications, and system message logging.

- An example adapter package named WmSampleAdapter.

  You can use this adapter as a model for developing your own adapters. This adapter enables you to exchange data with a simulated adapter resource provided with the example adapter. You will configure this adapter to perform a banking application. All of the underlying SampleAdapter class files are located in the *Integration Server_directory* \ packages\WmSampleAdapter directory. The section "Overview" on page 238 describes how the Sample Adapter was developed, and how you can configure and run it.

- A set of auxiliary Java services that you can use to replicate namespace nodes programatically and change the nodes' metadata appropriately when deploying an adapter to a different Integration Server.

- Online API Reference Javadoc files that provide detailed descriptions and usage information about all public APIs provided in the WmART package.

  The WmART package contains the components of the adapter run time as well as the ADK classes you extend to create the adapter implementation. The Javadoc files are located in the *Integration Server_directory* \doc\api\adk directory.

To create an adapter, you will need access to webMethods Integration Server (IS), Software AG Designer, a Java 1.3.1 compiler, and any Java editor.

Points of Integration

Before you build an adapter, determine which "point of integration" is most appropriate for integrating your back-end system. When determining this, consider such factors as the type and

volume of information you need to move between systems, and the number of systems you need to integrate. The major "point of integration" types include the following:

Data-Level Integration

An Enterprise Information System (EIS) that is integrated at the data level is one that moves data between data stores. That is, the data stores in your EIS are involved in processing or storing transaction data received from outside the EIS.

For example, you might need to expose a catalog and pricing database to customers to enable them to transact with it. Assuming that this database has a JDBC driver, you can use the database as your point of integration. That is, it would be the mechanism you use to build your adapter.

You might have several of your databases involved in an integration scheme. Suppose that in addition to the catalog and pricing database, you need to reference or update data located in another database to process a customer transaction. For example, suppose that after you extract data from an EIS-to-Integration Server purchase order (such as item, quantity, and price), you:

- Process that data using another database (perhaps you subtract the quantity ordered from the quantity in inventory, which is located in another database)
- Store the result (the updated quantity in inventory) in that other database.

Application Interface-Level Integration

An EIS that is integrated at the application interface level is one that leverages the exposed interfaces of custom or packaged applications, such as SAP, Siebel, and PeopleSoft applications. Such an interface, which usually consists of a set of the application's APIs, enables applications outside of your EIS to access business processes as well as simple information.

Method-Level Integration

An EIS that is integrated at the method level is similar to one that is integrated at the application interface level. Using this point of integration, you can enable any application to access the methods of any application in your EIS.

User Interface-Level Integration

An EIS that is integrated at the user interface level uses a more primitive, yet viable approach. This approach, also known as "screen scraping", leverages legacy EIS user interfaces as a common point of integration. For example, mainframe applications that do not provide database or business process level access may be accessed through the user interface of the application.

Development-Time Tasks and Support

At development time, you write the Java classes that define the adapter. To do this, you extend base classes provided by the ADK to produce an adapter definition class and template classes for connections, adapter services, polling notifications, and listener notifications.

At design time, adapter users will select these template classes to configure namespace nodes. A *namespace node* is a run-time component containing information about how a connection, adapter service, polling notifications, or listener notifications should behave at run time.

At development time, you will:

- "Create the Adapter Definition" on page 14.
- "Define Connections and Connection Factories" on page 15.
- "Define Adapter Service Templates" on page 15.
- "Define Polling Notification Templates" on page 15.
- "Define Listener Notification Templates" on page 15.
- "Define Metadata" on page 16.

Support for other development-time activities includes:

- "Transaction Support" on page 18.
- "Exception Support" on page 18.
- "Logging Support" on page 18.
- "Internationalization Support" on page 18.

Create the Adapter Definition

An adapter definition is the framework of an adapter. An adapter definition is recognized as an adapter by your Integration Server, but it lacks functionality. In later stages of development, you will add functionality by defining templates for adapter connections, adapter services, and optionally for polling notifications and listener notifications.

In the adapter definition, you create services and methods that:

- Describe the adapter to Integration Server.
- Describe the adapter's resources to Integration Server, including its connection factories, adapter service templates, polling notification templates, listener notification templates, and its default resource bundle implementation class.

  A *resource bundle* typically contains all display strings and messages used by the adapter at run time and at design time. A resource bundle is specific to particular locale. If you plan to run your adapter in multiple locales, you can include a resource bundle for each locale. Doing this enables you to internationalize an adapter quickly, without having to change any code in the adapter. Each adapter must provide at least a default resource bundle.
- Initialize properties and resources associated with the adapter, and clean up the resources when the adapter is disabled.
- Load the adapter onto Integration Server when the adapter is enabled, and unload the adapter when the adapter is disabled.

For more information, see Connections.

Define Connections and Connection Factories

A connection class functions as a connection to the resource with which the adapter must communicate (known as the *adapter resource*). The ADK provides a connection management service that dynamically manages connections and connection pools, based on rules that adapter users establish when they configure connections. To create connections, the connection management service uses a connection factory class.

Adapter users will create one or more connection namespace nodes, using the adapter's administrative interface. The creation of namespace nodes is a design-time activity. They will also create namespace nodes for adapter service templates and notification templates.

For more information, see Connections.

Define Adapter Service Templates

An adapter service defines an operation that the adapter will perform on a resource. You need to implement one adapter service template class for each resource operation the adapter will support.

For example, you might create a service template that fetches rows from a database based on a key field, and another service template that updates and inserts records into the database.

Adapter users will define their run-time adapter service nodes based on these templates. Designer provides facilities for creating, configuring, and testing adapter service nodes.

For more information, see "Adapter Services" on page 68.

Define Polling Notification Templates

A polling notification is a mechanism that tells your application when an event occurs in your adapter's resource. For example, your application might need to be notified when data is added, updated, and deleted from the resource. You need to implement one polling notification template class for each polling notification the adapter will support.

A polling notification periodically checks the resource at specified intervals for the occurrence of events, and publishes a document each time an event occurs in the resource.

Adapter users will define their run-time polling notification nodes based on these templates, in a way similar to how they configure adapter service nodes. Designer provides facilities for creating, configuring, and testing polling notification nodes.

For more information, see "Polling Notifications" on page 108.

Define Listener Notification Templates

A listener notification works in conjunction with a listener object to create a much more powerful model for detecting and processing events in the adapter resource than is possible with polling

notifications. You need to implement one listener notification template class for each listener notification the adapter will support.

A listener object is connected to an adapter resource, waiting for the server to deliver event notifications. The listener object is instantiated and is given a connection when the associated node is enabled. The listener object remains active with the same connection to monitor the resource activity until it is explicitly disabled.

When the listener detects a publishable event in the resource, it passes an object back to the server. The server will interrogate a configured list of listener notifications associated with the listener node until it finds a listener notification node that can process the event. The first listener notification to return true from this call will be invoked.

The ADK includes both a synchronous and an asynchronous processing model. An *asynchronous listener notification* publishes a document to a webMethods Broker queue, using the doNotify method. Adapter users may process the document's data any way they want to. For example, they can create an Integration Server trigger that receives the document and executes an IS service.

A *synchronous listener notification* invokes a specified IS service, and potentially receives a reply from the service and delivers the results back to the adapter resource. In this case, the notification object's runNotification method calls invokeService (instead of doNotify), to process the data produced by the notification. A synchronous listener notification does not publish a document.

For more information, see "Listener Notifications" on page 130.

Define Metadata

Connection factories, adapter service templates, polling notification templates, and listener notification templates all support a metadata interface. Metadata that you create in your adapter implementation primarily supports design-time activities. It describes parameters that the adapter users use to create namespace nodes for connections, adapter services, and notifications, and describes the data passed to and from adapter services and notification nodes.

When an adapter user creates a namespace node, the server interrogates the adapter for metadata to describe the parameter values supported by the node. The node stores the parameter values selected or entered by the user. These parameters are used to configure the appropriate implementation class when it is instantiated at run time.

Metadata parameter values are derived from the adapter's resource domain. A *resource domain* defines the domain of valid values for metadata parameters, based on rules and/or data that are specific to the adapter resource. A resource domain can have a name, a set of resource domain values, properties that affect the behavior of the resource domain, and associations with metadata parameters.

A resource domain can be either fixed or dynamic. A *fixed* resource domain displays default values that you provide for the resource domain parameters. With a *dynamic* resource domain, you use a method that enables the adapter to look up values for the parameters.

You can use resource domain values to constrain the values of parameters, to enable dynamic validation of user-supplied data, and to disable parameters, based on specific sets of values in

other parameters. A common use of resource domain values is to create a drop-down list of values for a parameter.

With both fixed and dynamic resource domains, you can allow adapter users to enter their own values. In addition, you can enable the adapter to validate these values using callbacks known as *adapter check values*.

With adapter services and polling notifications, resource domain values can interact with the Adapter Service Editor and the Adapter Notification Editor. As values in one parameter change, callbacks are made to the adapter to update resource domain values. In addition, the adapter can retrieve resource domain values directly from the adapter resource.

A parameter may contain a single value or an array of values. Parameters that hold arrays of values are called *sequence parameters*. The user interface for configuring connections does not allow you to populate more than the first element of a sequence parameter.

The metadata model provides the ability to:

- Define groups of parameters that will appear on different pages in the user interface.
- Define *resource domain lookups* that return drop-down lists of possible values for a parameter. Resource domain lookups may return values established at development time and/or retrieved at design time from the resource with which the adapter communicates.
- Establish sophisticated relationships between parameters using field maps, tuples, and resource domain dependencies.
    - *Field maps* are used to place sequence parameters in a grid-style table with each sequence parameter forming a column in the table.
    - *Tuples* are used in conjunction with field maps. When a set of parameters is placed in a tuple, resource domain lookups for those parameters are always made together. This is commonly used when columns are closely related, for example when column 1 contains field names, and column 2 contains data types for those fields.
    - *Resource domain dependencies* indicate that the values to be returned in a resource domain lookup are dependent on the value of one or more other parameters. Whenever the value of one parameter changes, if a second parameter's lookup is dependent on the value of the first, the lookup for the second parameter is performed again, with the new value of the first parameter being passed in as an argument to the lookup.
- Define signatures that define the data that should be passed to, or received from, the node when the service or notification is executed (typically by calling the execute method of the implementation class).
- Make interactive calls into the adapter to validate data entered by the user.
- Define tree structures to facilitate input of parameter values or to create logical groupings of fields in a signature.

Metadata is discussed in more detail throughout this document, in the chapters about connections, adapter services, and notifications.

Transaction Support

Integration Server considers a transaction to be one or more interactions with one or more resources that are treated as a single logical unit of work. The interactions within a transaction are either all committed or all rolled back. For example, if a transaction includes multiple database inserts, and one or more inserts fail, all inserts are rolled back.

Integration Server supports the following kinds of transactions:

- A *local transaction*, which is a transaction to a resource's local transaction mechanism
- An *XAResource transaction*, which is a transaction to a resource's XAResource transaction mechanism

Integration Server can automatically manage both kinds of transactions, without requiring the adapter user to do anything. Integration Server uses the container-managed (implicit) transaction management approach as defined by the JCA standard and also performs some additional connection management. This is because adapter services use connections to create transactions. However, there are cases where the adapter user needs to explicitly control the transactional units of work.

To support transactions, Integration Server relies on a built-in transaction manager. The transaction manager is responsible for beginning and ending transactions, maintaining a transaction context, enlisting newly connected resources into existing transactions, and ensuring that local and XAResource transactions are not combined in illegal ways.

For more information, see "Integration Server Transaction Support" on page 184.

Exception Support

The ADK provides standard exception classes that enable the adapter implementation to inform the server of exception conditions encountered by the adapter implementation. Any exception thrown from an adapter implementation will be caught and logged in the IS server and error logs. For details about the standard ADK exceptions, see the Javadoc for the com.wm.adk.error package.

Logging Support

The ADK provides the adapter with write access to the IS server and error logs. It also provides the ability to determine whether a particular message at a particular log level will be written to the log. All log messages generated by an adapter are stamped with the current date and time, as well as codes that uniquely identify the adapter generating the message. Logging services are tightly integrated with the resource bundle facilities for internationalization support. For more information, see the Javadoc for com.wm.adk.log.ARTLogger.

Internationalization Support

If you plan to run your adapter in multiple locales, you can internationalize an adapter quickly, without having to change any code in the adapter. To accomplish this, you use resource bundles.

A resource bundle typically contains all display strings and messages used by the adapter at run time and at design time. A resource bundle is specific to a particular locale.

For more information, see "Creating Resource Bundles" on page 36.

Design-Time Tasks

At design time, adapter users will configure and initialize the run-time components of the adapter. To do this, they will use the following kinds of Integration Server (IS) packages:

- The *adapter package*, which contains the adapter run-time components as well as the ADK classes that you, the adapter developer, extend to produce the adapter implementation.
- A *namespace node package*, in which adapter users will create the adapter's namespace nodes for connections, adapter services, polling notifications, and listener notifications.

  A *namespace node* (or *node*) is based on its corresponding Java classes. For example, a connection node is based on the Java connection classes created at development time.

Design-time tasks include the following:

- Create one or more namespace node packages.
- Create and initialize namespace nodes for connections, adapter services, polling notifications, listeners, and listener notifications. Adapter users can place all the nodes of an adapter in one package, or distribute them among multiple packages.
- Load the adapter package and namespace node packages into Integration Server.

To perform these tasks, adapter users require access to Integration Server, Designer, and a web browser.

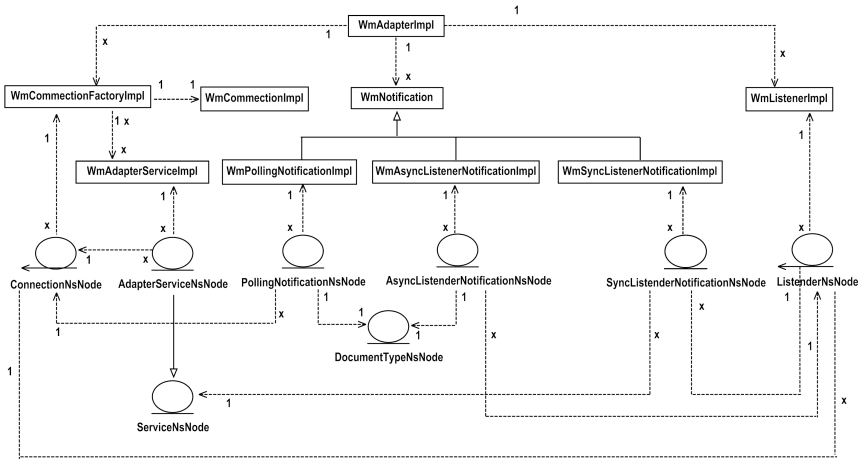
For more information, see "Design-Time Tasks" on page 150.

The Run-Time Conceptual Model

At run time, the adapter services and notifications communicate with the adapter resource to perform the function for which the adapter was created. Designer provides facilities for testing adapter services and notifications.

Although the following diagram is not strictly UML-compliant, it shows the relationships between the implementation classes of an adapter and their corresponding namespace nodes created at design time.

Relationships between adapter implementation classes and namespace nodes



In the diagram, entity icons represent namespace nodes, while standard class icons represent the implementation classes. The (unlabeled) dependency lines show direct references between classes and/or nodes. Thus, the adapter implementation class will directly reference supported connections or notification templates by directly referencing the connection factory and notification implementation class. The connection factory for each connection type references the service implementation class of each supported adapter service template.

Each namespace node depends on an implementation class. The implementation class for each node provides metadata that describes the data that is included in the node at design time. The node provides parameter settings that are passed back to the implementation class when that class is executed at run time. Note that service and notification nodes require a reference to a connection node that provides access to the resource.

For more information, see "Run Time Activities" on page 164.

2 Installing, Upgrading, and Uninstalling the Adapter Development Kit

- Overview 22
- Requirements 22
- Installing Adapter Development Kit 6.5 22
- Upgrading to Adapter Development Kit 6.5 23
- Uninstalling Adapter Development Kit 6.5 23

Overview

This chapter explains how to install, upgrade, and uninstall Adapter Development Kit 6.5. The instructions use Software AG Installer and Software AG Uninstaller. For complete information about other installation methods or installing other webMethods products, see the *Installing Software AG Products* guide for your release.

Requirements

For a list of the operating systems, webMethods Integration Server releases and Software AG Designer releases supported by the Adapter Development Kit, see the *webMethods Adapters System Requirements* .

Adapter Development Kit 6.5 has no hardware requirements beyond those of its host Integration Server.

Installing Adapter Development Kit 6.5

Before You Begin

1. If you are installing the Adapter Development Kit on an existing Integration Server, shut down the Integration Server.

2. Download the Software AG Installer from the Empower Product Support Web site.

Install Adapter Development Kit 6.5

To install Adapter Development Kit 6.5

1. Start the Installer wizard.

   - Choose the webMethods release that includes the Integration Server on which to install the Adapter Development Kit. For example, if you want to install the Adapter Development Kit on Integration Server 9.0, choose the 9.0 release.
   - If you are installing on an existing Integration Server, specify the webMethods installation directory that contains the host Integration Server. If you are installing both the host Integration Server and the Adapter Development Kit, specify the installation directory to use. Installer will install the components of the Adapter Development Kit in the *Integration Server_directory* as follows:

Component	Location
webMethods Adapter Development Kit Installation and User's Guide	*Integration Server_directory* \doc\adk directory
Adapter Development Kit API Reference (Javadocs)	*Integration Server_directory* \doc\api\adk directory
Sample Adapter (WmSampleAdapter package) and Sample Server for use with the Sample Adapter.	*Integration Server_directory* \packages directory

- In the product selection list, select **Adapters > webMethods Adapter Development Kit 6.5**.

2. To install documentation for Adapter Development Kit, on the Documentation panel in Installer, select **Adapter Readmes and Documentation**. Alternatively, you can download the adapter documentation at a later time from the Software AG Documentation Web site.

3. After installation is complete, start the host Integration Server.

Upgrading to Adapter Development Kit 6.5

You can upgrade to Adapter Development Kit 6.5 from Adapter Development Kit 6.0.1 or 6.1. Simply install Adapter Development Kit 6.5 over the existing installation using the instructions in "Installing Adapter Development Kit 6.5" on page 22.

Important:
Upgrading removes all installed Adapter Development Kit 6.0.1 or 6.1 components. You will not be able to restore the earlier version.

Uninstalling Adapter Development Kit 6.5

To uninstall Adapter Development Kit 6.5

1. Shut down the host Integration Server. You do not need to shut down any other webMethods products or applications that are running on your machine.

2. Software AG Uninstaller will not delete any user-defined Adapter Development Kit components such as connections, adapter services, or adapter notifications. Because these components will not work without the adapter, delete them manually using Designer, or Integration Server Administrator.

3. Start Uninstaller, selecting the webMethods installation directory that contains the host Integration Server. In the product selection list, select **Adapters > webMethods Adapter Development Kit**.

4. Restart the host Integration Server.

5. Uninstaller removes all Adapter Development Kit 6.5 related files that were installed into the Integration Server installation directory. However, Uninstaller will not delete the *Integration Server_directory* \packages\WmSampleAdapter directory if you added any files to it. You can delete this directory manually.

3 The Adapter Definition

- Overview 26
- Setting Up Your Environment for Adapters 27
- Adapter Definition Classes 30
- Adapter Definition Implementation Classes 30
- Creating a WmAdapter Implementation Class 31
- Creating Resource Bundles 36
- Deploying the Adapter 41

Overview

This chapter describes how to create an adapter definition. An adapter definition is the framework of an adapter. Although an adapter definition is recognized as an adapter by Integration Server, it lacks functionality. Other chapters describe how to add functionality by defining templates for connections, adapter services, polling notifications, and listener notifications.

To create an adapter definition, you perform the following tasks:

- Set up your environment for the adapter.

  You modify your classpath, create a package to contain the adapter, and register the adapter's major code with Software AG. For more information, see "Setting Up Your Environment for Adapters" on page 27.

- Create an adapter definition implementation class.

  This class, which must extend the base class WmAdapter, represents the main class of the adapter. In this class, you create services and methods that:

  - Describe the adapter to Integration Server.
  - Describe the adapter's resources to Integration Server, including its connection factories, notification templates, and its default resource bundle implementation class.
  - Initialize resources and properties referenced by the adapter definition when the adapter is enabled, and clean up the resources when the adapter is disabled (optional).
  - Load the adapter onto Integration Server when the adapter is enabled, and unload the adapter when the adapter is disabled.

  For more information, see "Creating a WmAdapter Implementation Class" on page 31.

- Create one or more resource bundles.

  This class, which must extend the base class java.util.ListResourceBundle, typically contains all display strings and messages used by the adapter at run time and at design time. A resource bundle is specific to a particular locale. If you plan to run your adapter in multiple locales, you can include a resource bundle for each locale. Doing this enables you to internationalize an adapter quickly, without having to change any code in the adapter. Each adapter must have one or more resource bundles. For more information, see "Creating Resource Bundles" on page 36.

- Deploy the adapter.

  You will:

  - Create startup and shutdown services.

    For more information, see "Creating Adapter Startup and Shutdown Services" on page 41.

  - Compile the adapter definition.

You compile your implementation class and construct the Java service nodes for your startup and shutdown services. The ADK provides a sample ANT script that you can run from the packages directory. For more information, see "Compiling the Adapter" on page 43.

- Configure the startup and shutdown services in Designer.

  For more information, see "Registering the Adapter Startup and Shutdown Services in Software AG Designer" on page 45.

Finally, the adapter user will use Integration Server Administrator to load the adapter by enabling the adapter package. These tasks are described in "Design-Time Tasks" on page 150.

Setting Up Your Environment for Adapters

To set up your environment, you must:

- Modify your classpath (see "Modifying Your Classpath" on page 27).
- Create an adapter package (see "Creating an Adapter Package" on page 28).
- Register your adapter's major code with Software AG, if required (see "Registering Your Adapter's Major Code with Software AG" on page 29).

Modifying Your Classpath

Including Jar Files in Your Classpath

To compile your source code, use jar files from the following locations:

Integration Server_directory \lib

Integration Server_directory \common\lib

Integration Server_directory \packages\WmART\code\jars

Integration Server_directory \packages\WmART\code\jars\static

where *Integration Server_directory* is the directory in which Integration Server is installed.

You would also need class files located in the *Integration Server_directory* \packages\WmART\code\classes directory. To use the class files, add the required folders from that location to your classpath, or package all folders in a jar and then add the jar to your classpath.

Specifying Your JDK in Your Classpath

You should specify JDK version 1.6 or higher in your classpath.

Modifying Your Classpath for Startup and Shutdown Services

As you will learn in "Creating Adapter Startup and Shutdown Services" on page 41, you must implement adapter startup and shutdown services as IS Java services. To do this, use Software AG Designer. Alternatively, you may implement these services as a Java class in your adapter, and use the jcode utility provided by Integration Server to convert them to IS Java services.

If you use Designer to do this, you must add wmart.jar to the compile classpath in *Integration Server_directory* /config/server.cnf. For example:

```
watt.server.compile=javac -classpath
{0};$IS_DIR\\packages\\WmART\\code\\jars\\wmart.jar; -d
{1} {2}
```

Creating an Adapter Package

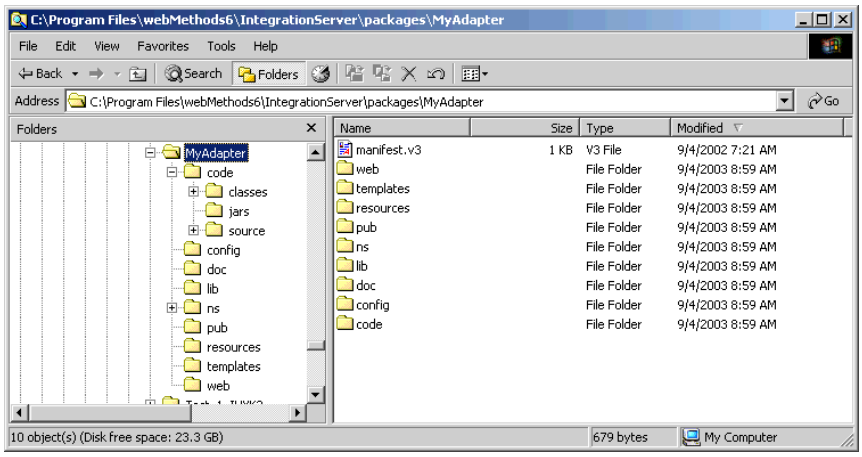
You create an adapter package in the same way you create any webMethods package, using Designer. If you need instructions for creating a package, see the *webMethods Service Development Help* for your release.

Important:
You must set the package dependency to the WmART package. You may set the version to the value *.*. The WmART package is automatically installed when you install Integration Server.

Designer creates a directory structure in which you can begin your adapter development, as shown below. Use the *adapterName*\code\source directory as your source base. Under the code\source directory, create directories corresponding to your Java package structure (for example, com\mycompany\adapter\myAdapter).

When you construct your build scripts, make sure they place your compiled code in the \code\classes directory.

IS package structure



Registering Your Adapter's Major Code with Software AG

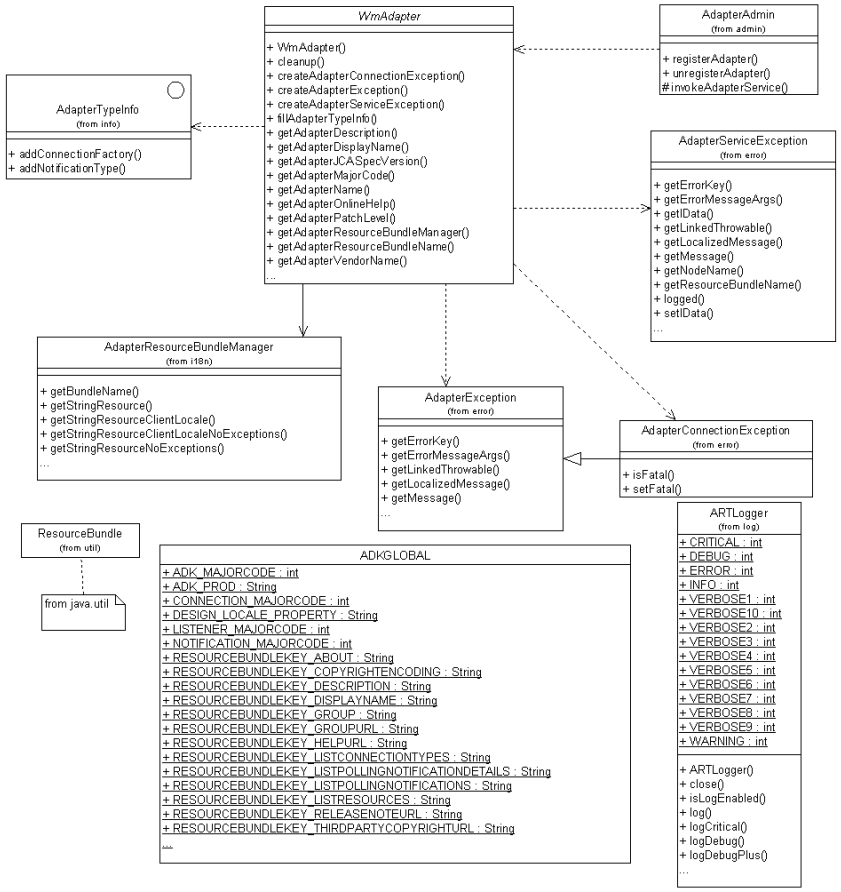
Every adapter built using the ADK requires an internal ID called a major code. A *major code* is an integer ID that Integration Server uses to distinguish journal log information between different adapter types. The major code is a four-digit number between 1 and 9999.

Each adapter implementation must have a major code that is unique from all other adapters built using the ADK that might be present in the same webMethods environment. Adapters with identical major codes will generate an error in the Integration Server log. More importantly, Integration Server log entries from same-code adapters will be indistinguishable.

The major code ranges are reserved as follows:

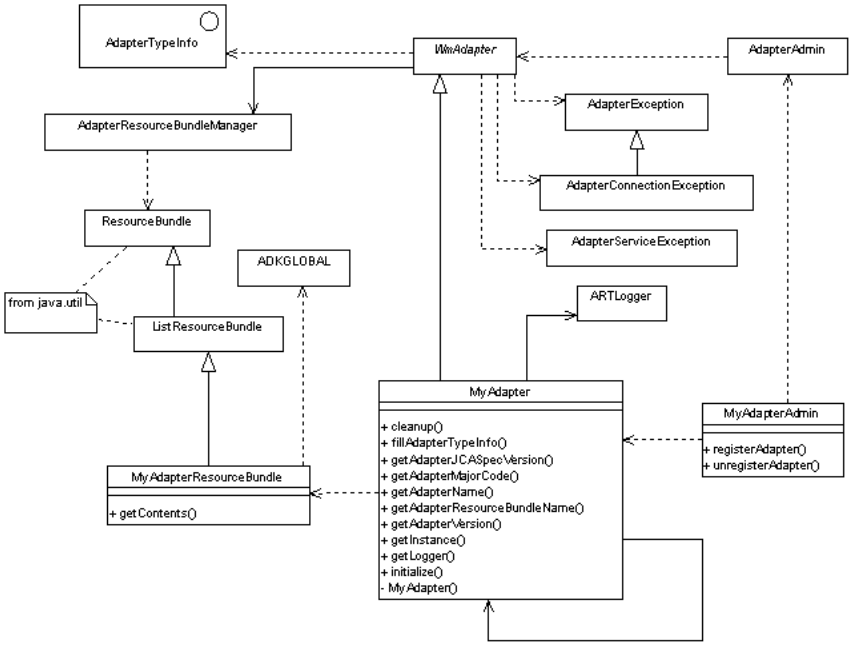
Major Code Range	Description
1-6999	This range is reserved for Software AG-built webMethods commercial adapters.
7000-8999	This range is reserved for adapters built by Software AG Development Partners. If you are a Development Partner, you must register your major code with Software AG. See "The Adapter Definition" on page 26.
9000-9999	This range is reserved for adapters you build for use within your own organization. You do not have to register the major codes for these adapters unless you will be running them in an environment where the adapters' major codes will conflict with other adapters in your webMethods environment.

Adapter Definition Classes



Adapter Definition Implementation Classes

The adapter definition implementation classes of MyAdapter



This chapter discusses the classes that implement the following classes:

- WmAdapter (see "Creating a WmAdapter Implementation Class" on page 31)
- ListResourceBundle (see "Creating Resource Bundles" on page 36)
- AdapterAdmin (see "Deploying the Adapter" on page 41)

Creating a WmAdapter Implementation Class

Before you begin, make sure you have created an adapter package to contain your adapter classes, as described in "Creating an Adapter Package" on page 28.

In your adapter package's main source code directory, you create an adapter definition by extending the WmAdapter base class. This class represents the main class of the adapter. In the example shown in "Adapter Definition Implementation Classes" on page 30, this class is MyAdapter. In this class, you create services and methods that:

- Describe the adapter to Integration Server (see "Describing the Adapter to Integration Server" on page 32).
- Describe the adapter's resources to Integration Server, including its connection factories, polling notification templates, listener notification templates, and its default resource bundle implementation class (see "Describing the Adapter's Resources to Integration Server" on page 32).
- Initialize resources and properties referenced by the adapter definition when the adapter is enabled, and clean up the resources when the adapter is disabled (optional; see "Initializing and Cleaning Up" on page 33).
- Add custom Dynamic Server Pages (DSPs) to your adapter's administrative interface (optional; see "Adding Custom DSPs to the Adapter Interface" on page 33).

The following sections describe the basic steps for implementing, deploying, and debugging an adapter definition. An example WmAdapter implementation class is presented in "Example WmAdapter Implementation Class" on page 34. Other chapters introduce more advanced techniques that make your adapter more manageable as you expand its functionality.

Describing the Adapter to Integration Server

To describe the adapter to Integration Server, you must override the following base class "get" methods in your WmAdapter implementation class:

Name	Description
getAdapterName	Returns the internal name of the adapter. This name is used to identify text fields in the resource bundle, and to identify the relationship between the adapter and its associated namespace nodes. For this reason, it is important that the value returned by this method does not change after namespace nodes have been created. This name must be unique within the scope of Integration Server.
getAdapterVersion	Returns the current version of the adapter. This value appears in the adapter's About page. This should not be confused with the package version used when setting package dependencies.
getAdapterJCASpecVersion	Returns the JCA standard version supported by the adapter. This value should always be "1.0".
getAdapterMajorCode	Must return a unique numeric value (if required) that you can obtain from Software AG (see "Registering Your Adapter's Major Code with Software AG" on page 29).

For examples of these methods, see "Example WmAdapter Implementation Class" on page 34.

Describe WmAdapter method

To register the adapter in adapter runtime for supporting parallel asset initialization, you must override the following base class method in your WmAdapter implementation

Name	Description
supportsParallelAssetInitialization	Indicates if the adapter supports parallel asset initialization. The default return value is false. Override this method and use the watt property for the adapter to enable or disable parallel asset initialization.

Describing the Adapter's Resources to Integration Server

The adapter resources that you must describe to Integration Server include:

- All connection factories and notification templates that the adapter supports.

  You set the names of all connection factory and notification implementation classes in an AdapterTypeInfo object in your adapter's implementation of the fillAdapterTypeInfo method. Set these names later, when you implement the connections and notifications, as described in:

  - "Updating AdapterTypeInfo" on page 62.
  - "Defining a WmPollingNotification Implementation Class" on page 116.
  - "Listener Implementation" on page 138.

- The default resource bundle implementation class.

  You deliver the name of your ListResourceBundle implementation class in the getAdapterResourceBundleName method. For information about resource bundles, see "Creating Resource Bundles" on page 36.

For examples of these methods, see "Example WmAdapter Implementation Class" on page 34.

Note:
When specifying any resource, you must use the fully qualified name of the resource's associated implementation class (rather than an object instance).

Initializing and Cleaning Up

In your WmAdapter implementation class, you can initialize resources and properties specific to your adapter when the adapter is enabled. For example, you might initialize the ArtLogger resource. (You can use an ARTLogger object to generate journal log entries for your adapter.) Because there can be only one ArtLogger instance per adapter (that is, per major code), you should manage the ArtLogger instance in your WmAdapter implementation class, but it is optional. You should use a static accessor method, which simply facilitates access to the ArtLogger instance. In the "Example WmAdapter Implementation Class" on page 34, this is accomplished using the getLogger method.

When the adapter is disabled, you must release the ArtLogger instance (and the adapter's major code) using the ArtLogger.close method. In the "Example WmAdapter Implementation Class" on page 34, this is accomplished using MyAdapter.cleanup, which is called from MyAdapterAdmin.unregisterAdapter.

You should release any other resources held by the adapter definition. Resources held by connections, adapter services, and notifications have their own cleanup mechanisms, as described in other chapters.

Adding Custom DSPs to the Adapter Interface

By default the adapter's administrative interface includes DSPs for connections, polling notifications, listeners, and listener notifications. You may add custom DSPs by overriding the following public methods provided in the WmAdapter abstract class:

Method	Description
getUiItemNames()	Returns a list of the label strings to insert in the left-hand panel of the Integration Server Administrator page for the adapter. Each label represents a link to the DSP to be executed.
getUiItemUrl (*String itemName*)	Returns the URL of the DSP associated with the given itemName. This effectively binds the displayable item link/label name with the DSP to launch when the adapter user selects that link. The URL is relative to the packages directory of the server installation. For example, if your adapter name is WmFoo and your DSP is bar.dsp, the URL would be \WmFoo\bar.dsp. In the file system, however, the .dsp file actually resides in packages\WmFoo\pub\bar.dsp; the pub directory is not included in the URL. You may also append arguments to the URL, using the standard notation `?=`.
getUiItemHelp (*String itemName*)	Returns the URL of the help file describing the custom DSP page associated with the given *itemName*. The pathname requirements are the same as for getUiItemUrl(). For example, if your help file is located in \WmFoo\pub\bar_dsp.html, the path returned by this method should be \WmFoo\bar_dsp.html.

The labels for these DSPs appear in the navigation area in the left-hand window of the interface, immediately below the default labels (the Connections, Polling Notifications, Listeners, and Listener Notifications labels) but above the About label.

For information about creating DSPs, see the document *Dynamic Server Pages and Output Templates Developer's Guide*.

Internationalization Considerations for Custom DSPS

You are responsible for implementing these methods (and the associated DSPs and help files) in a manner that takes into consideration the client's locale. In particular, the implementation of getUiItemNames() should perform the necessary resource domain lookups in order to return locale-specific values. The system will not automatically perform these lookups for you.

Example WmAdapter Implementation Class

Your WmAdapter implementation class must call the base class constructor (super()). The base class constructor calls several of the implementation class's "get" methods and instantiates your resource bundle (see "The Adapter Load Process" on page 46). Because these activities occur in the base class constructor after the first call to getInstance, it is vital that they do not invoke another call to getInstance. Doing this results in an endlessly recursive call to the constructor, which will ultimately crash the JVM and bring down Integration Server. In particular, pay attention to static initializers in your resource bundle. Because some of the resource bundles are keyed on the same string that is returned from WmAdapter.getAdapterName, do not populate that string in a static initializer, in a way similar to the following:

```
{MyAdapter.getInstance().getAdapterName() +
```

```
ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME, "My Adapter"}
```

This line of code in a static initializer of a resource bundle will produce the undesirable results described above.

The following example WmAdapter implementation class includes only the concepts discussed up to this point.

```
package com.mycompany.adapter.myadapter;
import com.wm.adk.WmAdapter;
import com.wm.adk.error.AdapterException;
import com.wm.adk.info.AdapterTypeInfo;
import com.wm.adk.log.ARTLogger;
import java.util.Locale;
```

```
public class MyAdapter extends WmAdapter
{
    public static final String ADAPTER_NAME = MyAdapter.class.getName();
    private static final int ADAPTER_MAJOR_CODE = 9001;
    private static MyAdapter _instance = null;
    private static ARTLogger _logger;
```

```
    private MyAdapter() throws AdapterException {super();}
    public String getAdapterName(){return ADAPTER_NAME;}
    public String getAdapterVersion(){return "1.0";}
    public String getAdapterJCASpecVersion(){return "1.0";}
    public String getAdapterResourceBundleName()
    {
        return MyAdapterResourceBundle.class.getName();
    }
    public int   getAdapterMajorCode(){return ADAPTER_MAJOR_CODE;}
    public static ARTLogger getLogger() {return _logger;}
    public static MyAdapter getInstance()
    {
        if (_instance != null)
            return _instance;
        else
        {
           synchronized (MyAdapter.class)
           {
               if (_instance != null)
               {
                   return _instance;
               }
               try
               {
                   _instance = new MyAdapter();
                   return _instance;
                }
                catch (Throwable t)
                {
                    t.printStackTrace();
                    return null;
                }
            }
        }
   }
```

```
   public void initialize() throws AdapterException
   {
       _logger = new ARTLogger(getAdapterMajorCode(),
                               getAdapterName(),
                               getAdapterResourceBundleName());
       getLogger().logDebug(9999,"My Adapter Initialized");
    }
    public void fillAdapterTypeInfo(AdapterTypeInfo info, Locale locale)
    {
    }
    public void cleanup() {
        if (_logger != null)
        _logger.close();
    }
}
```

Creating Resource Bundles

A resource bundle typically contains all display strings and messages used by the adapter at run time and at design time. A resource bundle is specific to particular locale. If you plan to run your adapter in multiple locales, you can include multiple, locale-specific resource bundles. Doing this enables you to internationalize an adapter quickly, without having to change any code in the adapter. An adapter must have one or more resource bundles.

A resource bundle consists of lookup keys that provide locale-specific objects (normally text strings). A lookup key consists of a constant provided by the ADKGLOBAL class (a class provided by the ADK's API) combined with your adapter's class name or a parameter name. For a list of these constants, see “Resource Bundle Lookup Keys” on page 38.

For example, the ADK uses the following lookup key whenever the display name of the adapter is required. (Assume that the method MyAdapter.getAdapterName will return the class name MyAdapter.)

```
MyAdapter.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME
```

The following lookup key produces a description for a parameter named password, which is used to configure a connection pool:

```
"password" + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION
```

For more examples, see “Example Resource Bundle Implementation Class” on page 41.

The adapter automatically references lookup keys to provide text for the following elements of your adapter:

- The display names, property descriptions, and online help links of the:
  - Adapter definition
  - Connection types
  - Adapter service templates
  - Polling notification templates

- Parameters used to configure nodes
- Metadata group names
- Log entries
- Exception text

Note:
You may also make explicit use of a resource bundle to localize other data, such as resource domain values, especially if those values are known at development time (for example, a list of known record status values). In these cases, the strategy for key composition is left to your discretion. For more information, see "Using Resource Bundles with Resource Domain Values" on page 180. Resource domain values are discussed in detail in "Resource Domains" on page 73.

To create a resource bundle implementation class

1. Extend the java.util.ListResourceBundle base class.

   In the example shown in "Adapter Definition Implementation Classes" on page 30, this class is MyAdapterResourceBundle. You should create your class in the same directory in which you created your WmAdapter implementation class.

   If you create multiple resource bundle classes, use the following naming convention:

   ```
   ResourceBundleName  ResourceBundleName_locale
   ```

   For example, a default resource bundle and a corresponding locale-specific bundle for use in Japan might be named:

   ```
   MyAdapterResourceBundle
   MyAdapterResourceBundle_ja
   ```

   For more information about resource bundle naming conventions, see the Javadoc for java.util.ResourceBundle.

   Note:
   You specify the adapter's default resource bundle in your WmAdapter implementation class (see "Describing the Adapter to Integration Server" on page 32). For each resource bundle lookup, the adapter uses the default resource bundle if a bundle specific to the target locale is not available.

2. In your subclass, create resource bundle lookup keys using the formats shown in "Resource Bundle Lookup Keys" on page 38.

3. In your WmAdapter implementation class, return the name of your default resource bundle using the getAdapterResourceBundleName method. For an example, see "Example WmAdapter Implementation Class" on page 34.

Resource Bundle Lookup Keys

The following table describes the automatic resource bundle lookups performed for each type of adapter element, and describes how the results are used.

Adapter Element	Key Format	Usage
Adapter definition	*adapterName* + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME	Required. Displays adapter name in Integration Server Administrator and in adapter interface.
	adapterName + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION	Required. Displays adapter description in adapter interface's About window.
	adapterName + ADKGLOBAL.RESOURCEBUNDLEKEY_ABOUT + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL	Required. Displays help hyperlink in adapter interface's About window.
	adapterName + ADKGLOBAL.RESOURCEBUNDLEKEY_LISTCONNECTIONTYPES + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL	Displays help hyperlink for connection type list in adapter interface when configuring new connection.
	adapterName + ADKGLOBAL.RESOURCEBUNDLEKEY_LISTPOLLINGNOTIFICATIONDETAILS + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL	Displays help hyperlink when modifying or viewing a polling notification's schedule.
	adapterName + ADKGLOBAL.RESOURCEBUNDLEKEY_LISTPOLLINGNOTIFICATIONS + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL	Displays help hyperlink for list of polling notifications.
Connection type	*ConnectionFactoryName*.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME	Displays connection type column in adapter interface.
	ConnectionFactoryName.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION	Displays description column in connection type listing when configuring connections.
	ConnectionFactoryName.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL	Displays help hyperlink when editing connection properties.
Adapter service template	*AdapterServiceName*.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME	Displays adapter service template name when selecting a template to create an adapter service in Designer.
	AdapterServiceName.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_ DESCRIPTION	Displays adapter service template description when selecting a

Adapter Element	Key Format	Usage
		template to create an adapter service in Designer.
	AdapterServiceName.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_ HELPURL	Displays help hyperlink when editing adapter service properties (focus must be in Properties window).
Polling notification template	*AdapterNotificationName*.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME	Displays template name column when selecting a template to create polling notification in Designer.
	AdapterNotificationName.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_ DESCRIPTION	Displays template description column when selecting a template to create polling notification in Designer.
	AdapterNotificationName.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_ HELPURL	Displays help hyperlink when editing polling notification properties (focus must be in Properties window).
Parameters used to configure nodes	*parameterName* + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME	Displays property display name when editing connection, adapter service, and polling notification properties in adapter interface and Designer.
	parameterName + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION	Displays tool tip when mouse is over parameter name when editing service and polling notification properties in Designer.
Metadata group names	*groupName* + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME	Displays property display name when editing parameter group for polling notification in Designer.
	groupName + ADKGLOBAL.RESOURCEBUNDLEKEY_GROUPURL	Overrides Designer help hyperlink on each tab associated with group.
Log entries	new Integer(minorCode).toString()	Text in all log repositories (server locale)
Exception text	new Integer(minorCode).toString()	Text in all log repositories (server locale)

* See "Considerations for Adapter Definition Lookup Keys" on page 40 and "Considerations for Specifying URLs in Resource Bundles" on page 40.

Note:
By default, if you omit a DISPLAYNAME lookup key in your resource bundle, the parameter or class name is used as the display name. Other omitted lookup keys display nothing.

Considerations for Adapter Definition Lookup Keys

As specified in the preceding table, the lookup keys specifying the adapter's display name, description, and vendor name are required. All other lookup keys are optional, but if you omit them, they may generate error messages in the log, and may make the adapter more difficult to use.

The *adapterName* reference in the Key Format column of the preceding table refers to the name of the adapter returned by *WmAdapterImpl*.getAdapterName (where *WmAdapterImpl* is the name of your WmAdapter implementation class). You may use the name of your WmAdapter implementation class if it is also returned by the getAdapterName method. Remember that the adapter name must be unique within the scope of Integration Server.

Alternatively, you may define a constant that both the lookup key and getAdapterName use. However, do not call *WmAdapterImpl*.getInstance.getAdapterName to retrieve the adapter name from either of these static initializers, or in the resource bundle constructor. Integration Server instantiates the resource bundle data during execution of the *WmAdapterImpl* super() constructor; calling getInstance from that point will produce undesirable results.

Considerations for Specifying URLs in Resource Bundles

Some lookup keys reference documents, such as help information and release note documents. If you include these document files with your adapter, place them under *AdapterPackageName*\pub (where *AdapterPackageName* is the file system directory under the Integration Server's packages directory where your adapter resides). For more information about the Integration Server file system structure, see "Creating an Adapter Package" on page 28.

In your lookup key, identify the resource file using the file path relative to the pub subdirectory. For example, to specify the location of the help file for SimpleConnection, a resource bundle would include the data value:

```
SimpleConnectionFactory.class.getName() +
ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL,
"\MyAdapter\help\eng\SimpleConnectionHelp.html"
```

This relative path is equivalent to the following path:

Integration Server_directory \packages\MyAdapter\pub\help\eng\SimpleConnectionHelp.html

This relative path scheme must be used for all URL references used by Designer as well as for the THIRDPARTYCOPYRIGHTURL clause. Other URL references accessed through the Server Administrator or the adapter's administrative interface may use other URL referencing schemes such as absolute paths or valid Internet addresses.

Example Resource Bundle Implementation Class

The following example creates lookup keys for each of the required resource bundle entries, plus an entry referencing an About page for the adapter. This About page reference assumes there is a file called AdapterAbout.txt in MyAdapter\pub. A production adapter would probably have a more complex directory structure under the pub subdirectory, and would include html files instead of simple text files.

```
package com.mycompany.adapter.myadapter;
```

```
import java.util.ListResourceBundle;
import com.wm.adk.ADKGLOBAL;
```

```
public class MyAdapterResourceBundle extends ListResourceBundle
{
     static final Object[][] contents  =
     {
          {MyAdapter.ADAPTER_NAME + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
                                    "My Adapter"}
          ,{MyAdapter.ADAPTER_NAME + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
                                    "Simple demonstration adapter"}
          ,{MyAdapter.ADAPTER_NAME + ADKGLOBAL.RESOURCEBUNDLEKEY_VENDORNAME,
                                    "My Company, Inc."}
          ,{MyAdapter.ADAPTER_NAME + ADKGLOBAL.RESOURCEBUNDLEKEY_ABOUT
                                   + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL,
                                    "MyAdapter/AdapterAbout.txt"}
          ,{"9999", "{0}"}               // general non-localized debug message
     };
```

```
     public Object[][] getContents() { return contents;}
```

```
}
```

Deploying the Adapter

Adapter users explicitly load and unload an adapter by enabling and disabling the adapter package, using Integration Server Administrator. These tasks are described in "Design-Time Tasks" on page 150. This section describes:

- "Creating Adapter Startup and Shutdown Services" on page 41.
- "Compiling the Adapter" on page 43.
- "Registering the Adapter Startup and Shutdown Services in Software AG Designer" on page 45.
- "The Adapter Load Process" on page 46.
- "The Adapter Unload Process" on page 47.

Creating Adapter Startup and Shutdown Services

To create adapter startup and shutdown services

1. Create a class in the same directory in which you created your WmAdapter implementation class.

   In the example shown in "Adapter Definition Implementation Classes" on page 30, this class is MyAdapterAdmin.

2. Create an adapter startup service as an IS Java service.

   The startup service must retrieve an instance of your WmAdapter implementation class and pass it to AdapterAdmin.registerAdapter.

   To implement a service as an IS Java service, use Designer. Alternatively, you may implement a service as a Java class in your adapter, and use the jcode utility provided by Integration Server to convert it to an IS Java service. For information about IS Java services and the jcode utility, see the *webMethods Service Development Help* for your release. For an example startup service that uses jcode, see "Example Adapter Startup and Shutdown Services" on page 42.

3. Create an adapter shutdown service as an IS Java service, in the same way you created your startup service.

   The shutdown service must:

   a. Retrieve an instance of your WmAdapter implementation class and pass it to AdapterAdmin.unregisterAdapter.

   b. Call any cleanup operations needed by your adapter. In most cases this is accomplished by calling MyAdapter.cleanup before the call to AdapterAdmin.unregisterAdapter. For more information about this cleanup method, see "Initializing and Cleaning Up" on page 33.

   For an example shutdown service that uses jcode, see "Example Adapter Startup and Shutdown Services" on page 42.

4. Compile the adapter, as described in "Compiling the Adapter" on page 43.

5. Register the adapter startup and shutdown services with Designer, as described in "Registering the Adapter Startup and Shutdown Services in Software AG Designer" on page 45.

Example Adapter Startup and Shutdown Services

The following example shows the startup and shutdown services used by My Adapter (with appropriate jcode tags):

```
package com.mycompany.adapter.myadapter;
import com.wm.adk.admin.AdapterAdmin;
import com.wm.data.IData;
import com.wm.app.b2b.server.ServiceException;
```

```
public class MyAdapterAdmin
{
    public static void registerAdapter(IData pipeline) throws ServiceException
    {
```

```
        // --- <<IS-START(registerAdapter)>> ---
        AdapterAdmin.registerAdapter(MyAdapter.getInstance());
        // --- <<IS-END>> ---
    }
```

```
    public static void unregisterAdapter(IData pipeline) throws ServiceException
    {
        // --- <<IS-START(unregisterAdapter)>> ---
        MyAdapter instance = MyAdapter.getInstance();
        instance.cleanup();
        AdapterAdmin.unregisterAdapter(instance);
        // --- <<IS-END>> ---
    }
}
```

Compiling the Adapter

Before you load your adapter, you must compile your implementation classes and construct the Java service nodes for your startup and shutdown services.

The following example code uses jcode. To see instructions for using jcode to generate Java services, see the *webMethods Service Development Help* for your release.

Note:
Make sure that you have modified your classpath as described in "Modifying Your Classpath" on page 27.

```
<?xml version="1.0"?>
<project default="classes">
    <property name="debug" value="on"/>
    <property name="optimize" value="off"/>
    <property name="deprecation" value="off"/>
    <property name="webM.home" value="../.."/>
    <!-- Installation directory of the webMethods -->
    <property name="server.home" value="${webM.home/IntegrationServer}"/>
    <!-- Directory of the IntegrationServer -->
    <property name="server.home" value="../.."/>
    <property="package" value = "MyAdapter" />  <!--needed for jcode -->
```

```
    <!-- classes belonging to this package -->
    <path id="this.package.classpath">
        <fileset dir=".">
            <include name="code/classes"/>
        </fileset>
    </path>
```

```
    <!-- All classes that need to be found by this script -->
    <path id="total.classpath">
        <pathelement
location="${server.home}/packages/WmART/code/jars/wmart.jar"/>
  <pathelement location="${server.home}/lib/wm-isserver.jar"/>
  <pathelement location="${server.home}/lib/wm-isclient.jar"/>
  <pathelement location="${webM.home}/common/lib/glassfish/gf.javax.resource.jar"/>
    <path refid="this.package.classpath"/>
    </path>
```

```
    <!-- Compile the java files of this package -->
```

```
    <target name="classes" depends="init">
        <mkdir dir="code/classes"/>
        <javac debug="${debug}" optimize="${optimize}"
        deprecation="${deprecation}" srcdir="code/source"
        destdir="code/classes">
            <classpath>
                <path refid="total.classpath"/>
            </classpath>
        </javac>
    <!-- needed for jcode -->
    <exec executable ="${server.home}/bin/jcode"
        vmlauncher = "false" failonerror = "true">
        <arg value = "fragall" />
        <arg value = "${package}" />
    </exec>
    </target>
```

```
     <!-- delete .class files built in this package -->
     <target name="cleanclasses">
         <mkdir dir="code/classes"/>
         <delete quiet="false">
             <fileset dir="code/classes" includes="**/*.class"/>
         </delete>
     </target>
```

```
     <!-- if this package depends on classes found in other packages,
     add targets to build those classes here. -->
     <target name="packageDependencies" depends=""/>
```

```
     <target name="init">
         <tstamp/>
     </target>
```

```
     <target name="all" depends="packageDependencies, classes" />
     <target name="clean" depends="cleanclasses"/>
     <target name="remake" depends="clean, packageDependencies, classes"/>
</project>
```

Debugging the Adapter

Perform the following to debug the adapter.

To debug an adapter

1. To use a debugger with your adapter, you must first shut down your Integration Server and start it using a custom start script.

2. Make a copy of your server.bat (or server.sh in a UNIX environment).

3. To edit your file copy to enable debugging, go to the line near the bottom of the file where the server is actually started. It should look similar to this:

```
1. rem run Integration Server
2. title webMethods Integration Server
3. %JAVA_RUN% -DWM_HOME="%WM_HOME%" -classpath %CLASSPATH%
```

```
   %IS_PROXY_MAIN% "%IS_DIR%"\bin\ini.cnf %PREPENDCLASSES_SWITCH%
   %PREPENDCLASSES% %APPENDCLASSES_SWITCH% %APPENDCLASSES%
   %ENV_CLASSPATH_SWITCH% %ENV_CLASSPATH% %1 %2 %3 %4 %5 %6 %7 %8 %9
```

4. Modify the file so that the debug settings are inserted immediately after the classpath, as follows:

```
1. set JDPA_ARGS=-Xdebug -Xnoagent -Djava.compiler=NONE
2. set JDWP_ARGS=
   -Xrunjdwp:transport=dt_socket,server=y,address=9999,suspend=n
3.
4. rem run integration server
5. title webMethods Integration Server
6. %JAVA_RUN% -DWM_HOME="%WM_HOME%" -classpath
   %CLASSPATH% %JDPA_ARGS%
   %JDWP_ARGS% %IS_PROXY_MAIN% "%IS_DIR%"\bin\ini.cnf
   %PREPENDCLASSES_SWITCH% %PREPENDCLASSES% %APPENDCLASSES_SWITCH%
   %APPENDCLASSES% %ENV_CLASSPATH_SWITCH% %ENV_CLASSPATH% %1 %2 %3 %4
   %5 %6 %7 %8 %9
```

5. Restart your server using the modified start script.

   You should be able to attach to Integration Server and "step" your code (assuming you compiled your code to include debug symbols (for example, javac -g)) using your favorite debugger. The following is a start script for connecting to Integration Server using JDB:

```
1.  rem @echo off
2.  set TARGET="C:\Program
    Files\webMethods6\IntegrationServer\packages\WmTest1Adapter\code\
    source"
3.
4.  set JAVA_MIN_MEM=64M
5.  set JAVA_MAX_MEM=64M
6.  set JAVA_MEMSET=-ms%JAVA_MIN_MEM% -mx%JAVA_MAX_MEM%
7.  set JAVA_DBG="C:\Program Files\Java\j2sdk1.3.1_08\bin\jdb"
8.  set ARGS=com.sun.jdi.SocketAttach:hostname=localhost,port=9999
9.
10. %JAVA_DBG% -sourcepath %TARGET% -connect %ARGS%
```

Registering the Adapter Startup and Shutdown Services in Software AG Designer

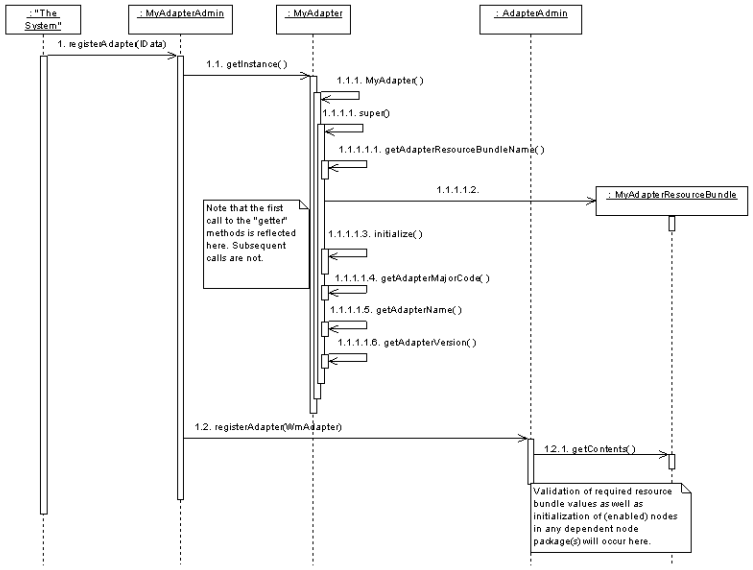
Before you register the adapter startup and shutdown services for the adapter package, make sure that you have:

- Created your adapter startup and shutdown services as described in "Creating Adapter Startup and Shutdown Services" on page 41.
- Successfully compiled the adapter as described in "Compiling the Adapter" on page 43.

Now, select your adapter package in Designer and identify your adapter package startup and shutdown services. If your startup and shutdown services do not appear in the adapter package, there has been an error either in compiling your code or in creating the Java services. Refer to the previous sections and correct the problem.

The Adapter Load Process

The adapter load process



Adapter users explicitly load and unload an adapter by enabling and disabling the adapter package, using Integration Server Administrator, as described in "Design-Time Tasks" on page 150. This figure shows how My Adapter loads the adapter into Integration Server at run time.

The loading process is described as follows:

1. The system calls MyAdapterAdmin.registerAdapter.

   This method is the designated package startup service; see "Creating Adapter Startup and Shutdown Services" on page 41.

2. The implementation of registerAdapter instantiates the MyAdapter class by calling its getInstance method. It then registers the MyAdapter object as an adapter by passing it to the registerAdapter method of the AdapterAdmin class provided by the ADK.

3. During the construction of the MyAdapter instance, super() calls the base class (WmAdapter) constructor, which instantiates the default resource bundle, calls MyAdapter.initialize, and retrieves basic information about the adapter. (The WmAdapter constructor only instantiates the default resource bundle; it instantiates and reads other resource bundles if a request is received using the locale of that supplementary resource bundle.)

4. The adapter load process is completed with a call to the static method AdapterAdmin.registerAdapter, which reads and evaluates the adapter's default resource bundle, and updates the Integration Server list of registered adapters. If the resource bundle does not provide the required resource bundle elements, the registration process fails, with an error log entry explaining the failure.

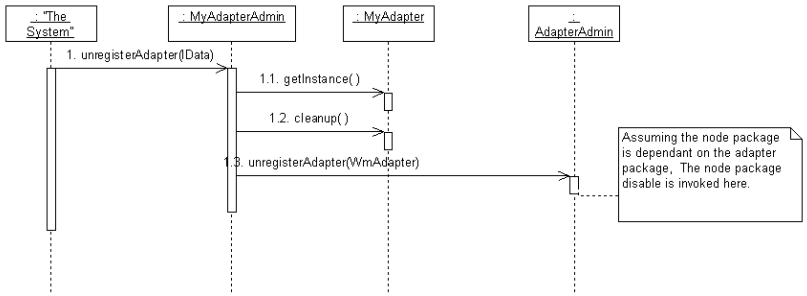
5. Upon completion of the adapter's registration, the adapter loads any dependant node packages. See the chapters about connections, adapter services, polling notifications, and listener notifications for descriptions of their respective load processes.

6. If the load was successful, the adapter name (as specified in the resource bundle) appears in the list of adapters in Integration Server Administrator. If it does not appear, refresh your browser page. If it still does not appear, check the log for errors. For information about debugging, see "Compiling the Adapter" on page 43.

The Adapter Unload Process

During server shutdown or package reload, the system disables the adapter's dependant nodes (or their packages) before it disables the adapter. When you explicitly disable an adapter's package using Integration Server Administrator, disable its dependant nodes first. The following diagram shows how MyAdapter unloads the adapter from Integration Server at run time.

The adapter unload process



To unload an adapter, the server calls MyAdapterAdmin.unregisterAdapter. This method is the designated package shutdown service; see "Creating Adapter Startup and Shutdown Services" on page 41. This method:

1. Retrieves the adapter instance.

2. Calls the adapter's cleanup method (if implemented).

3. Calls AdapterAdmin.unregisterAdapter.

Reporting Adapter Fix Levels

Once you deploy an adapter you may need to deliver a fix. You can identify the fix to Integration Server through the adapter's package manifest file. Whenever the adapter package is loaded, the "patch_history" field is read from the file and displayed on the adapter's About page. The management of this field is the responsibility of the adapter writer. The Integration Server facilities for creating partial package installations can be used deliver adapter fixes. For more information, see the *webMethods Integration Server Administrator's Guide* for your release.

4 Connections

- Overview 50
- Connection Classes 52
- Adapter Connection Implementation Classes 53
- Creating a WmManagedConnection Implementation Class 54
- Creating a WmManagedConnectionFactory Implementation Class 56
- Updating the Resource Bundle 62
- Updating AdapterTypeInfo 62
- Connection Class Interactions 62
- Configuring and Testing Connection Nodes 66

Overview

An adapter connection connects an adapter to an adapter resource. This chapter describes the classes provided by the ADK to support connections, and how adapters support them.

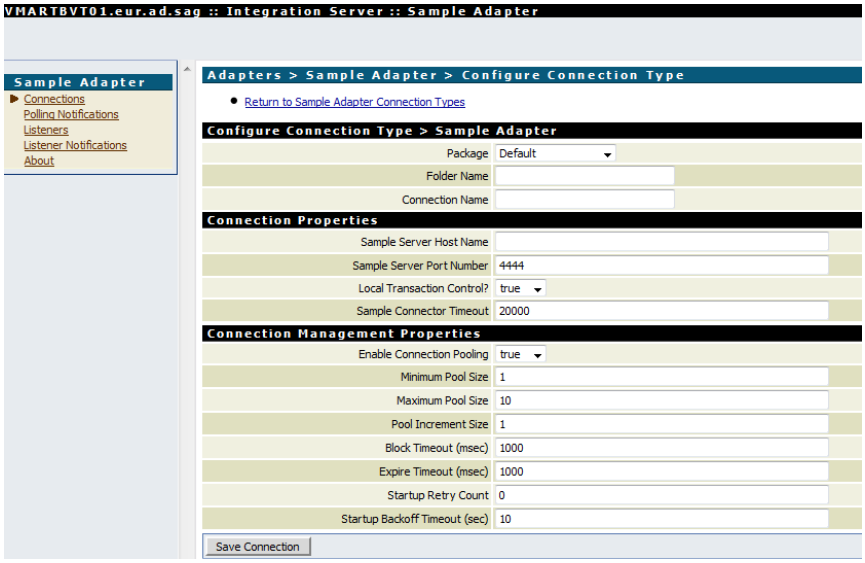
Connection Factories

The ADK's adapter connection model uses a factory method pattern in which a connection factory object is responsible for creating connection objects. In many cases, both the factory and its connections will wrap comparable functionality provided in the resource's libraries. For example, an adapter connection factory may wrap a data source class provided by a database vendor, from which it creates database connections and wraps them in an adapter connection object produced by the factory.

Each connection factory in an adapter implementation constitutes a *connection type* on the adapter's administrative interface. You can define one or more connection types for an adapter. If you need a different set of configuration parameters, create another connection type. For example, if you have a kind of request that requires special security requirements, create a separate connection type for it.

A connection factory is also responsible for defining implementation-specific parameters and for making them available to the connection. The adapter uses these parameters at design time, when adapter users create connection namespace nodes. For example, note the following connection type configuration window of the Sample Adapter.

The Configure Connection Type window of the Sample Adapter



Connection Management Properties

Each field in the Connection Properties section shown in the preceding figure has a corresponding metadata parameter provided by the associated connection factory. At run time, the node passes

the adapter user's settings for these fields to the connection factory, which makes them available to each connection the factory creates.

Two of these fields pertain to initializing the connection pool at startup:

- **Startup Retry Count** specifies the number of times that the system should attempt to initialize the connection pool at startup if the initial attempt fails, before issuing an AdapterConnectionException.
- **Startup Backoff Timeout** specifies the number of seconds to wait between each attempt to initialize the connection pool.

These fields provide flexibility in managing connections in environments where network anomalies are commonplace. They are significant in the following situations: when a new connection is enabled; when Integration Server starts; and when the package containing a (previously) enabled connection node is reloaded.

For information about other connection management parameters, see "Configuring Connection Nodes" on page 153.

Connection Management

The server includes a connection management service that dynamically manages connections and connection pools based on the settings stored in the connection namespace node (such as the connection pooling and timeout fields in "Connection Management Properties" on page 50).

When a connection namespace node is enabled, the server uses the connection factory to initialize the pool, creating a number of connection instances equal to the minimum configured pool size. Whenever a connection is needed by an adapter service or notification, the ADK provides a connection from the pool. If no connections are available in the pool, and the maximum pool size has not been reached, a new connection is retrieved from the connection factory. If the pool is full, the requesting thread will block the amount indicated in the Block Timeout field until a connection becomes available. For information about configuring your connection pool, see "Configuring and Testing Connection Nodes" on page 66.

Overview of Creating Connections

To implement a connection, you perform the following steps:

- Create a WmManagedConnection implementation class.

  This class is primarily responsible for wrapping the connection to your resource. The method that instantiates the class receives information from the connection factory. Accessing the resource is your responsibility. For more information, see "Creating a WmManagedConnection Implementation Class" on page 54.

- Create a WmManagedConnectionFactory implementation class, which:
  - Constructs a new connection object.
  - Identifies all adapter service templates supported by the factory's connections.

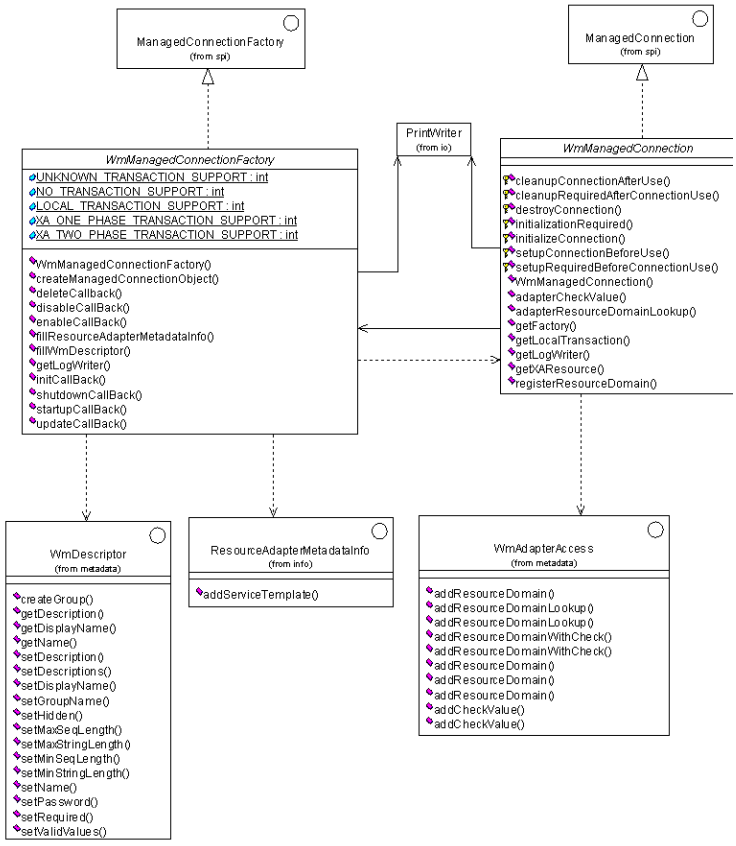
- Specifies the transactional capabilities of the factory's connections.
- Creates webMethods metadata for the connection factory.

For more information, see "Creating a WmManagedConnectionFactory Implementation Class" on page 56.

- Update the adapter's resource bundle with display names and other display-oriented data for the connection implementation and its parameters. For more information, see "Updating the Resource Bundle" on page 62.
- Link the connection type to the adapter as described in "Updating AdapterTypeInfo" on page 62.
- Compile and reload your adapter, and test the connection as described in "Configuring and Testing Connection Nodes" on page 66.

Connection Classes

ADK adapter connection classes

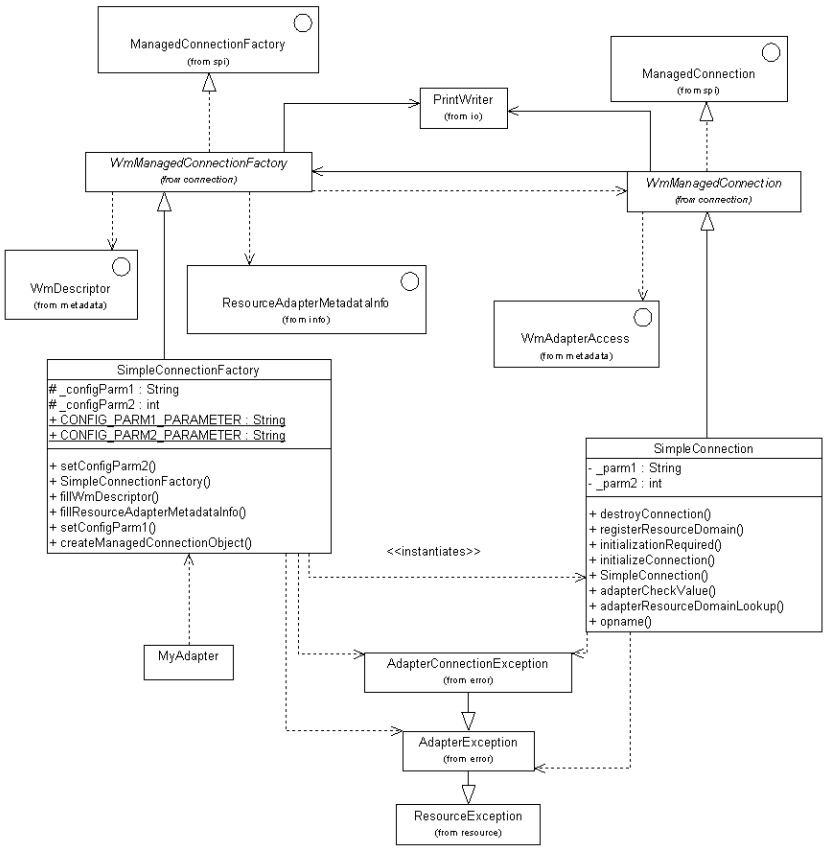


The ADK's base connection classes are dependent on the following external interfaces. These interfaces are used as arguments of the abstract methods that you must override in the connection implementation classes:

Interface	Description
WmDescriptor	Creates webMethods metadata (see "The WmDescriptor Interface" on page 60).
ResourceAdapterMetadataInfo	Specifies adapter services supported by the connection. This interface is an extended version of javax.resource.cci.ResourceAdapterMetadata. For more information, see "Creating a WmManagedConnectionFactory Implementation Class" on page 56.
WmAdapterAccess	Used in conjunction with metadata operations for associated adapter services and notifications. For more information, see the Javadoc.

Adapter Connection Implementation Classes

Adapter connection implementation classes



The diagram above shows how a typical adapter implements the connection classes.

At design time, each connection factory in an adapter implementation constitutes a *connection type* on the adapter's administrative interface. For a connection factory to be recognized as part of the adapter, you must specify the class in the fillAdapterTypeInfo method in the adapter's WmAdapter implementation class (see "Updating AdapterTypeInfo" on page 62).

To learn how the server uses connection classes at design time to support the creation and management of connection namespace nodes, see "Connection Class Interactions" on page 62.

Note:
The connection implementation classes will often use both the AdapterException class and the AdapterConnectionException class. The main difference between these classes is the impact on the connection pool (see "Receiving AdapterConnectionExceptions" on page 66).

Creating a WmManagedConnection Implementation Class

You create a connection by extending the abstract base class WmManagedConnection. In the example shown in "Adapter Connection Implementation Classes" on page 53, this class is SimpleConnection.

The WmManagedConnection implementation class is primarily responsible for wrapping the connection to your resource. The method in the connection factory that instantiates this class receives information from the connection factory. Accessing the resource is your responsibility.

In this class, you override the following methods:

Method	Description
adapterCheckValue	Supports metadata for associated services and notifications.
adapterResourceDomainLookup	Supports metadata for associated services and notifications.
destroyConnection	This method is called when the connection is removed from the pool, freeing implementation-specific resources.
initializationRequired	This method should return True if initializeConnection is present in the implementation class.
initializeConnection	This method is called once if initializationRequired returns True. Optional.
registerResourceDomain	Supports metadata for associated services and notifications.

For examples of these methods, see "Example WmManagedConnection Implementation Class" on page 55. (The initializeConnection and initializationRequired methods are not used in this example) For more information, see the Javadoc.

Example WmManagedConnection Implementation Class

The following example is very simple; it reflects no interaction with an adapter resource. Instead of showing a wrapped connection to an adapter resource, this WmManagedConnection implementation only prints debug messages to indicate that it has been created. The connection receives the appropriate parameter information from its factory. The constructor arguments pass values for the metadata parameters from the factory to the individual connection.

Note:
The methods that support metadata for services and notifications are described in "Adapter Services" on page 68, and "Polling Notifications" on page 108.

```
package com.mycompany.adapter.myadapter.connections;
```

```
import com.wm.adk.connection.WmManagedConnection;
import com.wm.adk.metadata.*;
```

```
import com.mycompany.adapter.myadapter.MyAdapter;
```

```
public class SimpleConnection extends WmManagedConnection
{
    String _parm1;
    int _parm2;
    public SimpleConnection(String configParm1, int configParm2)
    {
        super();
        _parm1 = configParm1;
        _parm2 = configParm2;
        MyAdapter.getLogger().logDebug(9999,
                                "Simple Connection created with parm1 = "
                                + _parm1 + "and parm2 = " +
                                Integer.toString(_parm2));
    }
    public void destroyConnection()
    {
        MyAdapter.getLogger().logDebug(9999,"Simple Connection
        Destroyed");
    }
//  The remaining methods support metadata for related services, etc.
//  Implement content as needed.
    public void registerResourceDomain(WmAdapterAccess access)
    {
    }

    public Boolean adapterCheckValue(   String serviceName,
                                        String resourceDomainName,
                                        String[][] values,
                                        String testValue)
    {
        return null;
    }
    public  ResourceDomainValues[] adapterResourceDomainLookup(String
        serviceName, String resourceDomainName, String[][] values)
    {
        return null;
    }
}
```

Important:
You can make a callback to the factory using WmManagedConnection.getFactory. If you do this, do not call the parameter "set" methods on the factory. Doing this will produce unpredictable results.

Creating a WmManagedConnectionFactory Implementation Class

You create a connection factory by extending the abstract base class WmManagedConnectionFactory. In the example shown in "Adapter Connection Implementation Classes" on page 53, this class is SimpleConnectionFactory.

In this class, you override the following methods:

Method	Description
createManagedConnectionObject	Constructs a new connection object (for example, SimpleConnection).
queryTransactionSupportLevel	Specifies the transactional capabilities of this factory's connections. For more information, see this method's Javadoc.
fillWmDescriptor	Supports webMethods metadata constructs. For more information, see "The WmDescriptor Interface" on page 60.
deleteCallBack	Called when a connection factory is deleted.
disableCallBack	Called when a connection factory is disabled.
enableCallBack	Called when a connection factory is enabled.
initCallBack	Called when a connection factory is initialized.
startupCallBack	Called when a connection factory is started.
shutdownCallBack	Called when a connection factory is stopped.
updateCallBack	Called when a connection factory is updated.

For more information about Connection Callbacks, see "Connection Callbacks" on page 57. For examples of these methods, see "Example WmManagedConnectionFactory Implementation Class" on page 60.

The following sections describe the basics of the metadata model for connection factories.

Connection Callbacks

The WmManagedConnectionFactory base class defines a set of callback methods that you can override in any connection factory implementation class. These callbacks are called during state changes on the connection node.

The following table describes which methods are called on an enabled connection for certain operations.

If the user...	The enabled connection receives these callbacks:
Enables a disabled connection	enableCallBack startupCallBack
Disables an enabled connection	shutdownCallBack disableCallBack
Enables a package containing an enabled connection	initCallBack startupCallBack
Disables a package containing an enabled connection	shutdownCallBack
Deletes a package containing an enabled connection	shutdownCallBack
Reloads a package containing an enabled connection	shutdownCallBack initCallBack startupCallBack
Starts Integration Server	initCallBack startupCallBack **Note:** This only occurs if the Connections are contained in a package that is enabled.
Shuts down Integration Server	shutdownCallBack **Note:** This only occurs if the Connections are contained in a package that is enabled

The following table describes which methods are called on a disabled connection for certain operations.

If the user...	The disabled connection receives these callbacks:
Creates a connection	initCallBack
Updates a connection	updateCallBack
Deletes a connection	deleteCallBack
Copies a connection	initCallBack (on the new connection)
Enables a package containing a disabled connection	initCallBack
Reloads a package containing a disabled connection	initCallBack
Starts Integration Server	initCallBack **Note:** This only occurs if the Connections are contained in a package that is enabled

Note:
If more then one callback is listed, they occur in the specified order from top to bottom. If an action is not listed in the table then no callback will occur.

For more information, refer to the Javadoc for the WmManagedConnectionFactory class.

webMethods Metadata Parameters

Each adapter interface field for configuring connection types should have a corresponding metadata parameter, provided by the associated connection factory. The webMethods metadata models are designed to define, refine, organize, and constrain parameters used to configure namespace nodes for connections, adapter services, and notifications. The metadata model for connections is the foundation for the more complicated metadata model used to configure adapter services and notifications.

Each metadata parameter is identified by a "set" accessor method. For example:

```
public void setServerName(String name);
```

```
public void setPortNumber(int portNumber);
```

Metadata Parameter Names

You derive the name of a parameter from the name of its "set" method. Typically, you remove the prefix (set) and make the first letter lower case. Thus, the "set" methods above would define metadata parameters named serverName and portNumber. For example:

```
public void setServerName(String name) {
}
```

```
public void setPortNumber(int portNumber) {
}
```

The complete naming convention rules follow the Java bean property naming conventions. For more information, see http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

Some naming variations include:

```
setFoo()  -> parameter name is "foo"
setfoo()  -> parameter name is "foo"
setFOO()  -> parameter name is "FOO"
setFOo()  -> parameter name is "Foo"
set_foo() -> parameter name is "_foo"
set_Foo() -> parameter name is "_Foo"
```

Metadata Parameter Arguments

A metadata parameter's "set" method should accept a single argument, whose data type will define the data type of the parameter. For connections, this type should be limited to a Java primitive or a java.lang. String. Arrays (or sequence parameters) are allowed, but the adapter's administrative interface only provides widgets to access the first element of the array. Other object types are interpreted as external Java bean classes, as described in "Implementing Metadata Parameters Using External Classes" on page 176.

In addition to defining the parameter name and data type, the "set" method of a metadata parameter is used at run time to pass the value of the parameter to the connection factory. The examples shown in this section show metadata accessor methods as methods of the connection factory implementation class. In fact, any method in the connection factory implementation conforming to the naming convention will be interpreted as a metadata parameter accessor method.

Metadata Parameter get Accessor Methods

A metadata parameter may have a corresponding "get" accessor method (following the same naming convention) that returns the same data type as the argument of the "set" method. The adapter uses these "get" methods only to retrieve default parameter values when creating a new connection.

Creating a "get" method with a different data type than its corresponding "set" method results in an error. In addition, creating a "get" method without a "set" method will produce a parameter whose values are unusable.

Note:
All namespace nodes store parameter settings based on the parameter name. If you delete or change the name of your accessor methods, the parameter names stored in the namespace nodes associated with that class will no longer be valid. From that point forward, any use of that node (at design time or at run time) will fail. If you no longer need a metadata parameter after upgrading a deployed adapter, hide the parameter (using WmDescriptor.setHidden) instead of deleting it.

The WmDescriptor Interface

The WmDescriptor interface controls how metadata parameters are handled at design time, during data entry.

To create webMethods metadata for your connection factory, you use the WmDescriptor interface within the WmManagedConnectionFactory.fillWmDescriptor method. Following are some commonly used WmDescriptor methods. For examples, see "Example WmManagedConnectionFactory Implementation Class" on page 60.

Use this method ...	To ...
createGroup	Specify the order in which parameters should appear on the adapter's administrative interface. Group names are not displayed.
setDescriptions	Set display names and online help links for the current connection type. This call queries your resource bundle for information about the connections, its parameters, and its groups. The ResourceBundleManager argument must be identical to WmAdapter.getResourceBundleManager.
setDisplayName	Assign a more user friendly parameter name. Alternatively, use setDescriptions to use a localized display name.
setMinStringLength setMaxStringLength setRequired	Control adapter user input.
setPassword	Display asterisks when adapter users enter passwords. This method also displays a confirmation field in which users must re-enter passwords. Only one copy of the parameter value will be set in the connection factory. Do not use setPassword on more than one parameter per WmDescriptor.
setValidValues	Create a drop-down list of values for a parameter.

Most WmDescriptor methods apply to a specified metadata parameter. In these cases, the name of the parameter must be passed as a String.

Example WmManagedConnectionFactory Implementation Class

The example connection factory, SimpleConnectionFactory, supports some simple metadata parameters. Note the following:

- Two constant strings represent the names of the metadata parameters for the connection factory. These strings must correspond to the names of the "set" accessor methods in this class. Using

string constants is not required, but they make the code more manageable because metadata methods and the resource bundle often contain multiple references to the parameter name.

- Each parameter name has a "set" method and an attribute to hold the value passed to the "set" method. This implementation does not provide "get" methods because there are no default values for parameters (and this adapter has no other use for the "get" methods).
- The createManagedConnectionObject method simply instantiates the connection class, passing the parameter values in the connection constructor.
- The fillWmDescriptor method uses the createGroup method to determine the order in which the parameters will appear on the display. The last line of fillWmDescriptor causes the server to retrieve display text for the connection from the example resource bundle. (To see the updated resource bundle, see "Updating the Resource Bundle" on page 62.)
- The final method in the example, fillResourceAdapterMetadataInfo, indicates which services are supported by the connection.

```
package com.mycompany.adapter.myadapter.connections;
import com.mycompany.ad   apter.myadapter.*;
```

```
import com.wm.adk.connection.*;
import com.wm.adk.info.ResourceAdapterMetadataInfo;
import com.wm.adk.metadata.WmDescriptor;
import com.wm.adk.erro r . AdapterException;
import java.util.Locale;
```

```
public class Si mp le ConnectionFactory extends  Wm Man agedConnectionFactory
{
    public static final String CONFIG_PARM1_PARAMETER = "configParm1";
    public static final String CONFIG_PARM2_PARAMETER = "configParm2";
    private String _configParm1;
    private int _configParm2;
    public void setConfigParm1(String val){_configParm1 = val;}
    public void setConfigParm2(int val){_configParm2 = val;}
    public SimpleConnectionFactory(){super();}
    public WmManagedConnection
        createManagedConnectionObject(javax.security.auth.Subject subject,
        javax.resource.spi.ConnectionRequestInfo cxRequestInfo)
    {
        return new SimpleConnection(_configParm1, _configParm2);
    }
    public void fillWmDescriptor(WmDescriptor d,Locale l) throws
              AdapterException
    {
        d.createGroup("nameNotUsedForConnections",
                                        new String[]{CONFIG_PARM2_PARAMETER,
                                       CONFIG_PARM1_PARAMETER});
        d.setValidValues(CONFIG_PARM2_PARAMETER, new String[] {"1","2","4"});
       d.setRequired(CONFIG_PARM1_PARAMETER);
       d.setDescriptions(
                 MyAdapter.getInstance().getAdapterResourceBundleManager(),l);
    }
    public void fillResourceAdapterMetadataInfo(ResourceAdapterMetadataInfo
info,
        Locale locale)
    {
    }
```

```
}
```

Updating the Resource Bundle

Notice in "Example WmManagedConnectionFactory Implementation Class" on page 60 that the call to WmDescriptor.setDescriptions in the fillWmDescriptor implementation causes the server to look for display names and other display data in the adapter's resource bundle. (For more information about resource bundles, see "Creating Resource Bundles" on page 36.)

To provide this information, the resource bundle is updated with entries for the Simple Connection implementation and its parameters as follows:

```
,{SimpleConnectionFactory.class.getName()
                              + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
                              "Simple Connection"}
,{SimpleConnectionFactory.class.getName()
                              + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
                              "Simple framework for demonstration purposes"}
,{SimpleConnectionFactory.class.getName()
                              + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL,
                              "MyAdapter/SimpleConnectionHelp.txt"}
,{SimpleConnectionFactory.CONFIG_PARM1_PARAMETER
                              + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
                              "String Setting"}
,{SimpleConnectionFactory.CONFIG_PARM2_PARAMETER
                              + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
                              "Integer Setting"}
```

Updating AdapterTypeInfo

The final step for creating a connection is to link the connection type to the adapter by updating your fillAdapterTypeInfo method in your WmAdapter implementation class. For example, MyAdapter accomplishes this as follows:

```
import com.mycompany.adapter.myadapter.connections.*;
.
.
.
public void fillAdapterTypeInfo(AdapterTypeInfo info, Locale locale)
{
    info.addConnectionFactory(SimpleConnectionFactory.class.getName());
    info.addNotificationType(SimpleNotification.class.getName());
}
```

Connection Class Interactions

The server uses connection classes at design time to support the creation and management of connection namespace nodes (as well as adapter service and notification namespace nodes). At run time, the ADK connection manager creates and releases connections as necessary, based on the pool configuration in the connection node and the demand for access to the adapter resource. This section describes the interactions of connection classes during the management of connection namespace nodes, as well as the basic flow used whenever connections are created or destroyed.

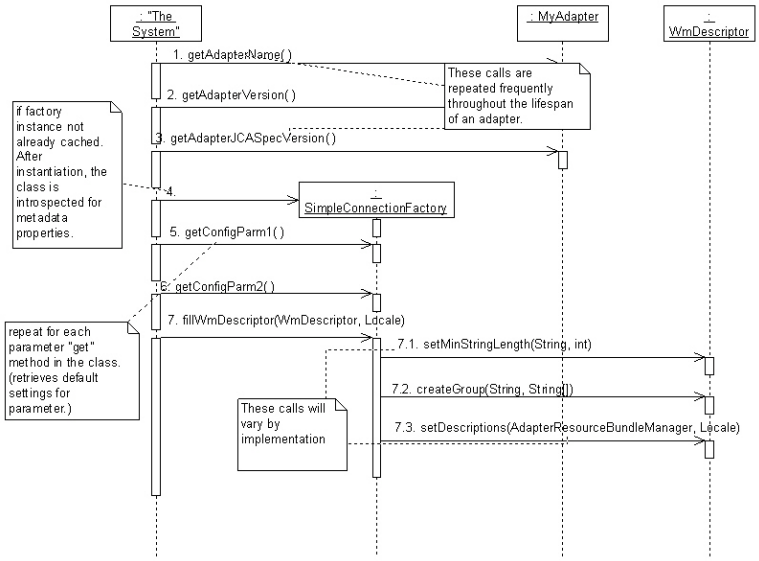
The specifics of how these activities pertain to the interactions of adapter services and notifications are described later in this document.

Retrieving Connection Metadata

When an adapter user creates or edits a connection node, the server interrogates the connection factory implementation to retrieve metadata as follows:

1. Gets the adapter name, version, and JCA version (steps 1-3 in the following figure).
2. Instantiates the connection factory if it is not already cached, and introspects the connection factory for metadata parameters (step 4). The server retrieves the "set" accessor methods described in "webMethods Metadata Parameters" on page 58.
3. Calls the "get" method for each metadata parameter (if present) and retrieves any default values that may be provided for those parameters (steps 5-6). (While "get" methods may have other uses in the implementation, this is their only use from the ADK's perspective.)
4. Calls the fillWmDescriptor method of the connection factory (step 7), which can specify the order in which parameters should appear on the adapter's administrative interface, set display names and online help links, and constrain adapter user input, as described in "The WmDescriptor Interface" on page 60.

Retrieving connection metadata



Enabling Connection Nodes

Connection nodes are disabled by default; users must explicitly enable them using the adapter's administrative interface (as described in "Configuring and Testing Connection Nodes" on page 66). If a connection node is enabled when the server shuts down, it will also be enabled at server startup.

The server performs the following actions to enable a connection node:

1. Obtains a connection factory instance.

   If the connection manager has already created a connection factory instance (and it is cached for use with this node), that instance is used; otherwise, the manager instantiates a new instance, as shown in step 1 in the following figure.

2. Updates the connection factory instance with the metadata parameter settings using the "set" methods described in "webMethods Metadata Parameters" on page 58.

3. Calls the enableCallBack method to call any adapter specific operations for the enable state change (step 4 in the following figure).

4. Calls the queryTransactionSupportLevel method to get the transaction support capabilities of connections created by this factory (step 5 in the following figure).

   Transaction support depends on the capabilities of the adapter resource and the current metadata parameter settings. This method determines whether connections from this node can participate in a transaction that may involve other connections, adapters, or resources. For more information about transaction support, see "Specifying Transaction Support in Connections" on page 195.

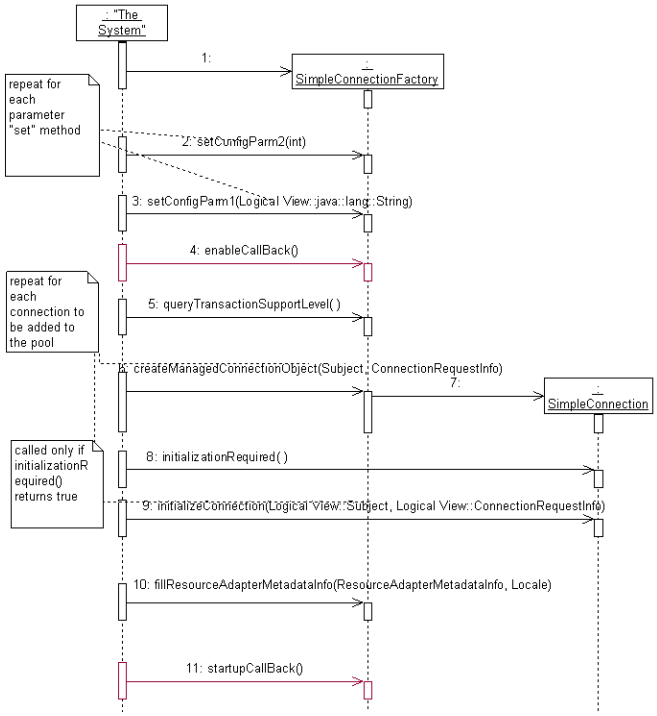
5. Initializes the connection pool.

   For each connection it will place in the pool (based on the minimum pool size specified when the node was created, as described in "Configuring and Testing Connection Nodes" on page 66), the server calls the createManagedConnectionObject on the connection factory and the initialization methods on the resulting connection object (steps 6 through 9 in the following figure). If connection pooling is disabled, a single connection will be created, initialized, and then destroyed.

6. The server calls the fillResourceAdapterMetadataInfo method on the connection factory to register the types of adapter service templates supported by the connection.

7. Calls the startupCallBack method to perform any adapter specific operations for the startup state change (step 11 in the following figure).

Enabling a connection node



Creating Connections

The connection manager requests new connections from an adapter's connection factory when the node is enabled, and whenever there is demand for a connection and the pooling limits defined by the connection node have not been exceeded. If pooling for the node is disabled, the manager creates a connection for each request, and destroys it when the request is completed. The process of creating a connection is shown in steps 5-7 in the previous figure.

Disabling Connection Nodes

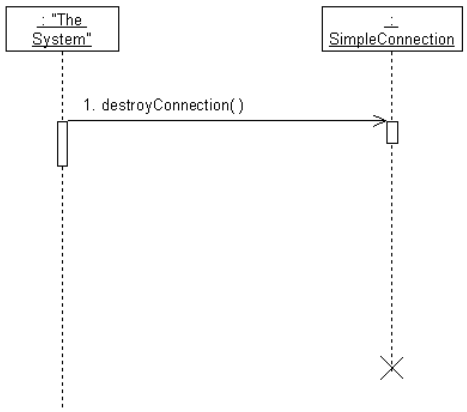
Disabling a connection node causes the connection manager to release all connections in the connection pool, and to reject any further requests for connections from that node. The connection manager destroys connections that are currently being used by an adapter service or notification when the current invocation of the adapter service or notification is completed. To see how a connection is released, see the figure in "Releasing Connections" on page 66.

Note:
A listener instance holds the connection it retrieves during listener initialization for the lifetime of the listener instance. Disabling the connection node will not impact this connection already held by the listener, but it will prevent the listener from starting or restarting in the event of an AdapterConnectionException. For more information, see "Listener and Listener Notification Interactions" on page 135.

Releasing Connections

When a connection is not in use and is not needed to maintain a connection pool, the connection manager calls the connection implementation's destroyConnect method and removes all references to the object, allowing it to be garbage collected.

Releasing a connection



Receiving AdapterConnectionExceptions

When the server receives an AdapterConnectionException thrown from an adapter, the server re-sets the connection node associated with the exception. This means that the connections in the pool are destroyed and not recreated until the next connection request.

Configuring and Testing Connection Nodes

Now you are ready to configure a connection node and verify that it establishes a connection to your adapter resource as follows:

- Compile your adapter as described in "Compiling the Adapter" on page 43.
- Reload your adapter as described in "Loading, Reloading, and Unloading Packages" on page 152.
- Configure and enable a connection node as described in "Configuring Connection Nodes" on page 153.

5 Adapter Services

- Overview 68
- Adapter Service Classes 68
- The Metadata Model for Adapter Services 70
- Adapter Service Template Interactions 89
- Adapter Service Implementation 92
- Configuring and Testing Adapter Service Nodes 106

Overview

An adapter service defines an operation that the adapter will perform on an adapter resource. Adapter services operate like webMethods Integration Server flow services or Java services. Adapter services have input and output signatures, you call them within flow services, and you can audit them from the Integration Server's audit system.

Like a connection, an adapter service consists of a Java class component and a namespace node in which design time settings are stored in metadata parameters. Adapter services support the same basic metadata constructs supported by connections, as well as:

- Additional data types.
- More sophisticated widgets.
- The ability to define the signature of the adapter service node.

  This means that adapter users can specify what data to search for in the flow service pipeline when the adapter service is called, and what data to place in the pipeline during execution of the service. Using these signatures, you can link adapter services to other Integration Server elements as part of a total integration solution.

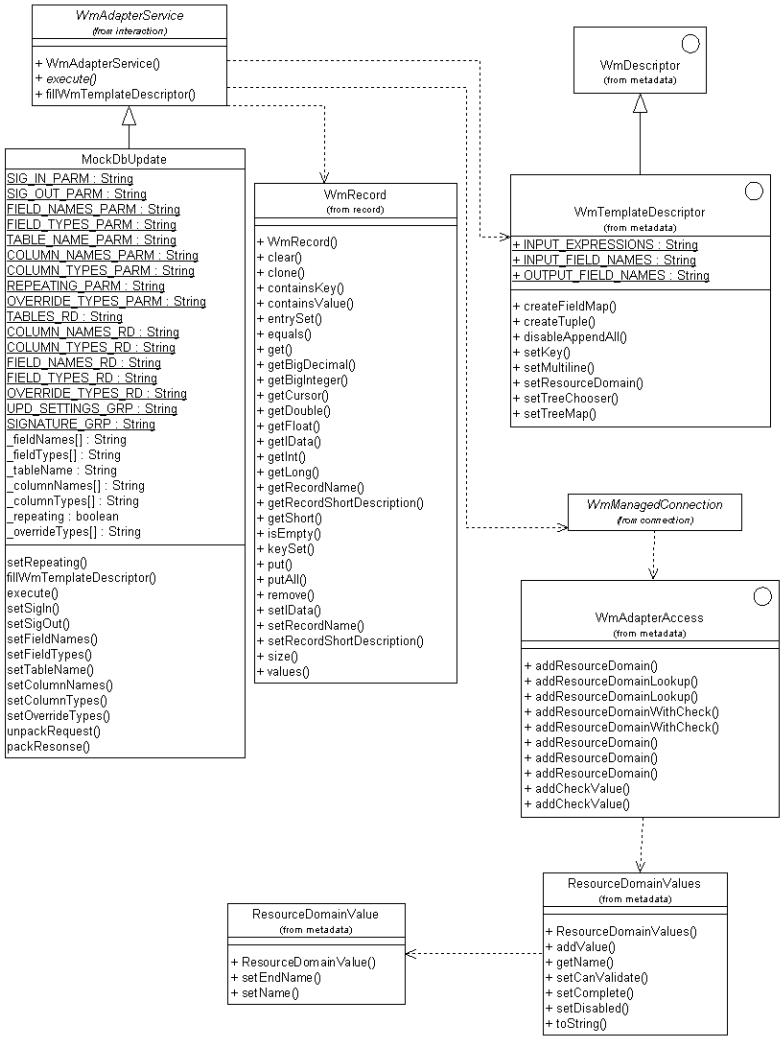
Software AG Designer provides facilities for adapter users to create, configure, and test adapter service nodes.

Adapter Service Classes

You create an adapter service template by extending the base class WmAdapterService. Because most adapters include several adapter service template classes, it is common to place them in their own package.

The following figure shows the classes provided by the ADK to support adapter service templates. It also shows the WmAdapterService implementation class MockDbUpdate.

ADK adapter service classes and the implementation class MockDbUpdate



In your WmAdapterService implementation class, override the following methods:

Method	Description
execute	Receives a WmManagedConnection object from the adapter implementation, and a WmRecord containing the pipeline data. **Note:** This method is abstract in the base class, so failing to override it will produce a compilation error. For more information, see "Adapter Service Execution" on page 87.
fillWmTemplateDescriptor	Serves to modify how metadata parameters are handled during data entry. Failing to override fillWmTemplateDescriptor results in a run-time error. For more information, see "The WmTemplateDescriptor Interface" on page 70.

Method	Description
metadataVersion	Returns the current version of the metadata. **Note:** Override this method if the template has multiple metadata versions.
fieldsToIgnoreInMetadataDefinition	Uses metadata version as input and returns an array of the fields that are not applicable to the metadata version provided. **Note:** Override this method if the template has multiple metadata versions. Failing to override fieldsToIgnoreInMetadataDefinition results in breaking old services.

Registering Adapter Service Templates in Connection Factories

You must register each adapter service template class in the connection factory classes for the connection types that will support the services. You do this by passing the class name to the ResourceAdapterMetadataInfo.addServiceTemplate method from the fillResourceAdapterMetadataInfo method of the connection factory. An example is shown in "Defining a WmAdapterService Implementation Class" on page 93.

The Metadata Model for Adapter Services

The following sections describe the basics of the metadata model for adapter services.

Metadata Parameters for Adapter Services

Metadata parameters for adapter services use the same model that connections use (as described in "webMethods Metadata Parameters" on page 58). However, the restriction on sequence parameters (arrays) does not apply to adapter services. The Adapter Service Editor supports widgets that allow adapter users to view and manipulate array values in several ways, as described in the next section.

It is important to note that providing default values to parameters through a "get" method is not as valuable in the context of adapter services. This functionality is largely replaced by the use of resource domains. Some functionality, such as the specification of the run time signature of the service, requires values to be provided through the resource domain facilities. For more information, see "Resource Domains" on page 73.

The WmTemplateDescriptor Interface

The WmTemplateDescriptor interface extends the WmDescriptor interface (which is described in "The WmDescriptor Interface" on page 60). Like WmDescriptor, WmTemplateDescriptor controls

how metadata parameters are displayed, and defines rules for data entry. To populate WmTemplateDescriptor, you include a fillWmTemplateDescriptor method in your adapter service.

Important:
Do not call the base class version of the method (by calling super()).

WmTemplateDescriptor introduces new methods and concepts, which are described in "The WmTemplateDescriptor Methods" on page 72. In addition, it modifies the behavior of some of the methods inherited from WmDescriptor, as described below:

- createGroup

  Specifies the order in which parameters should appear in Designer. Unlike with connections, you can create multiple groups for adapter services. Each group will correspond to a tab in the Adapter Service Editor, with the group name becoming the key from which its display name is identified. Each group has a resource bundle entry that provides the display name shown on the screen. Group names are only displayed if you do not provide a display name in your resource bundle.

  Parameters can be assigned to only one group. If some or all of a service's metadata parameters are not assigned to a group, a default group is created and they are assigned to it. This is true even if the parameter is hidden. Each group may have a display name and a help URL specified in the resource bundle.

- setValidValues

  Use the resource domain facilities (recommended) instead of this method. For more information, see "Resource Domains" on page 73.

- setPassword

  Displays asterisks when adapter users enter passwords.

  Note:
  The Adapter Service Editor does not support the automated password confirmation facilities described for connections.

- setDescriptions

  Searches the resource bundle for display names, descriptions, help URLs for the service, metadata parameters, and any groups that have been created at the point setDescriptions is called. Since you normally want group display fields to be loaded from the resource bundle, you typically call setDescriptions at the end of the fillWmTemplateDescriptor implementation.

- setHidden

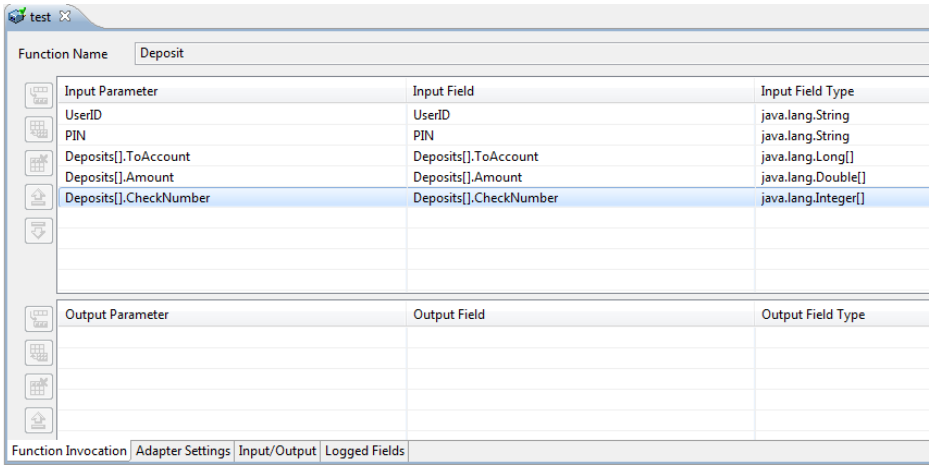
  Designates properties as hidden. A hidden property is not displayed in the Adapter Service Editor.

Note:
Some of these methods have implicit order requirements because the methods build on activities performed in previous method calls (see "The WmTemplateDescriptor Methods" on page 72).

By default, the Adapter Service Editor displays each metadata parameter in its own widget, based on the parameter's data type. These widgets include:

Widget	Data Type
Text box	String parameter
Table widget	Array
Check box	Boolean
A text box with scrollable values	Numeric

Adapter Service Editor: an adapter service



The WmTemplateDescriptor Methods

To modify and enhance the default behavior of WmTemplateDescriptor, use the following WmTemplateDescriptor methods:

- createFieldMap

  Use this method to organize parameters into columns of a single table widget. For more information, see "Field Maps" on page 81.

- createTuple

  This method is another "grouping" mechanism that you can use to modify the behavior of a resource domain lookup, and how resource domain values are applied in a field map. For more information, see "Tuples" on page 82.

- setMultiline

  Changes the widget from a standard text box to a multi-line text box. This widget also supports text import from files. The resulting parameter value may include embedded returns.

- setResourceDomain

Associates a metadata parameter of an adapter service with a resource domain supported by the service's connection. For more information, see "Resource Domains" on page 73 and "Associating Metadata Parameters with Resource Domains" on page 76.

Some of these methods have implicit order requirements because they build on activities performed in previous methods calls. For example, call setDescriptions after the final call to createGroup so that the resource bundle lookups can include all groups defined in the service. Use the calls in the following order:

1. createGroup
2. createFieldMap
3. createTuple
4. setResourceDomain
5. setDescriptions

Note:
If you create tuples, make setResourceDomain calls for tuple parameters in the order in which the parameters are set in the tuple.

Resource Domains

A *resource domain* defines the domain of valid values for metadata parameters, based on rules and/or data that are specific to the adapter resource. Resource domains can have properties that affect the behavior of the resource domain, and associations with metadata parameters.

You can use resource domain values to:

- Assign default values for parameters.
- Enable the adapter to look up parameter values in the adapter resource.
- Enable adapter users to supply their own parameter values, and enable the adapter to validate these values.
- Disable parameters, based on specified sets of values in other parameters.

A common use of resource domain values is to create a drop-down list of data values for a parameter, much like the WmDescriptor.setValidValues method in connection factories. However, resource domain values differ from the setValidValues lists in two important ways:

- Resource domain values can interact with the Adapter Service Editor.

  As values in one parameter change, callbacks are made to the adapter to update resource domain values. For example, if parameter A contains a list of database table names, and parameter B contains a list of column names, then when a table is selected in parameter A, the resource domain values used in parameter B can be updated to reflect the columns from the table selected in parameter A.

- Resource domain values can be retrieved directly from the adapter resource, using a WmManagedConnection instance.

To create a resource domain, you perform the following tasks:

1. Register the resource domain and its properties in your WmManagedConnection implementation (see "Registering Resource Domains" on page 74).
2. Associate the adapter's metadata parameters with the resource domain (see "Associating Metadata Parameters with Resource Domains" on page 76).
3. Populate the resource domain with values in your WmManagedConnection implementation (see "Populating Resource Domains with Values" on page 77).

Registering Resource Domains

You register resource domains in your WmManagedConnection implementation using the WmManagedConnection.registerResourceDomain method.This method has a single argument of the WmAdapterAccess type. For more information, see"Creating a WmManagedConnection Implementation Class" on page 54.

Registering a resource domain name establishes the definition of the resource domain within a given scope. That is, you can register one resource domain to be used by all adapter services that use the connection, or you can register multiple resource domains on a service-by-service basis. You must register the name of each resource domain supported by the connection (and by extension, each resource domain used by any service supported by the connection).

In the registerResourceDomain method, you:

- Create a ResourceDomainValues object, which represents a list of values for a resource domain (see "Populating Resource Domains with Values" on page 77).
- Specify whether the resource domain is fixed or dynamic.

  A *fixed* resource domain displays default values that you provide for the resource domain parameters; use the WmAdapterAccess.addResourceDomain method (see "The addResourceDomain Method" on page 75).

  With a *dynamic* resource domain, you use the addResourceDomainLookup method to enable the adapter to look up values for the parameters, based on changes to dependency parameters (see "The addResourceDomainLookup Method" on page 75).
- Specify whether adapter users can provide their own parameter values, and enable the adapter to validate these values.

  With both fixed and dynamic resource domains, you may allow adapter users to enter their own values, allowing them to add to the current list of values for the resource domain. In addition, you can enable the adapter to validate these values, using callbacks known as *adapter check values*. To do this, you use the following methods:

  - ResourceDomainValues.setComplete(false) indicates that adapter users can enter their own parameter values; it sets a flag named complete

- ResourceDomainValues.setCanValidate(true) indicates that the adapterCheckValue method will validate user-supplied parameter values; it sets the canValidate flag
- WmAdapterAccess.addCheckValue calls the WmManagedConnection.adapterCheckValue method; see "Adapter Check Value Callbacks" on page 80.

  Note:
  addCheckValue must appear after addResourceDomain or addResourceDomainLookup.

"Example of Registering Resource Domains" on page 75 illustrates how to use these methods.

The addResourceDomain Method

You use the addResourceDomain method to define a fixed resource domain with one or more values. This method expects one or more ResourceDomainValues objects. For more information about resource domain values and their settings, see "Populating Resource Domains with Values" on page 77.

The addResourceDomainLookup Method

You use the addResourceDomainLookup method to define a dynamic resource domain. This method supplies a reference to an object that the Adapter Service Editor will use to make callbacks when the adapter service node is being configured. Resource domain lookups can only be performed against WmManagedConnection objects, so the "this" reference is generally used as the object reference argument. For example:

```
access.addResourceDomainLookup("aSampleResourceDomainName", this);
```

For more information, see "Resource Domain Lookups" on page 78.

Example of Registering Resource Domains

The following example registers two resource domains:

```
101. public void registerResourceDomain(WmAdapterAccess access)
102.    throws AdapterException
103. {
104.    ResourceDomainValues tableRdvs = new ResourceDomainValues(
105.       MockDbUpdate.TABLES_RD, mockTableNames);
106.    tableRdvs.setComplete(true);
107.    access.addResourceDomain(tableRdvs);
108.
109.    access.addResourceDomainLookup(MockDbUpdate.COLUMN_NAMES_RD,this);
110.    access.addResourceDomainLookup(MockDbUpdate.COLUMN_TYPES_RD,this);
111.
112.    ResourceDomainValues rdvs = new ResourceDomainValues(
113.       MockDbUpdate.OVERRIDE_TYPES_RD, new String[] {""});
114.    rdvs.setComplete(false);
115.    rdvs.setCanValidate(true);
116.    access.addResourceDomain(rdvs);
117.    access.addCheckValue(MockDbUpdate.OVERRIDE_TYPES_RD,this);
...
121. }
```

In this example, note that:

- Lines 104-105 create a ResourceDomainValues object (identified by the constant MockDbUpdate.TABLES_RD).
- Line 106 indicates that adapter users may not supply their own values to the resource domain.
- Line 107 adds a resource domain to this object.
- Lines 109-110 adds lookups for this object, indicating that the resource domain is *dynamic*. For these lookups, you must pass an instance of the connection that can satisfy the lookup. This is accomplished by using the "this" reference.
- Lines 112-113 create an empty ResourceDomainValues object.
- Line 114-115 indicates that adapter users may supply their own values, and the adapter will validate these values. setCanValidate(true) calls the adapterCheckValue method (line 117) for each value that is not already in the resource domain.
- Line 116 adds a resource domain to this object. This is a fixed resource domain because no lookups are performed.

Associating Metadata Parameters with Resource Domains

The WmTemplateDescriptor interface provides setResourceDomain methods that you use to assign the metadata parameters of an adapter service to a resource domain supported by the service's connection.

The primary form of this method is:

```
void setResourceDomain(String name,
                       String resourceDomainName,
                       String[] dependencies)
```

where:

Argument	Description
name	The name of the parameter being assigned (parameter names are described in "webMethods Metadata Parameters" on page 58).
resourceDomainName	A name that matches the name registered in the connection, as described in "Registering Resource Domains" on page 74.
dependencies	A list of any other metadata parameter names in the current adapter service upon which the value of the parameter in the first argument depends (see "Parameter Dependencies" on page 77).

Parameter Dependencies

Dependencies are important to dynamic resource domain lookups. When an adapter user changes the value of a parameter in the dependency list, a lookup retrieves a new set of resource domain values.

For example, for the parameters named columns and tables, you might assign the columns parameter to a resource domain called columnsLookup, with a dependency on the tables parameter as follows:

```
d.setResourceDomain("columns", "columnsLookup", new String[] {"tables"});
```

When the tables parameter changes, WmManagedConnection.adapterResourceDomainLookup determines the new value or values that will be applied to the columns parameter (as described in "Resource Domain Lookups" on page 78). Depending on the properties of a resource domain, the lookup may be used to set the value of a parameter, or to provide a list of possibilities from which the adapter user may select a value.

Variant forms of setResourceDomain, including the concept of useColumns, are described in "The useParam Argument of setResourceDomain" on page 144.

Populating Resource Domains with Values

To populate a fixed or dynamic resource domain with values, you implement the ResourceDomainValues class in your WmManagedConnection implementation.

This class is the primary container for controlling the behavior of adapter service parameters. It is used during the registration process when you register a fixed resource domain, and it is returned from the adapterResourceDomainLookup method in response to a dynamic callback (as described in "Resource Domain Lookups" on page 78).

A ResourceDomainValues object contains:

- The name of the resource domain.
- The current list of values for the resource domain.

  This list can contain an array of either strings or ResourceDomainValue objects. This array constitutes the list of values that appears in the appropriate widgets in the Adapter Service Editor. The displayed values are communicated as strings, regardless of the data type of the parameter associated with the resource domain.

  Note:
  Using a values list that cannot be converted to the parameter's data type results in a run-time error. Null and empty strings are not accepted in numeric parameters.

- Methods that:
  - Allow adapter users to supply values (set the setComplete method to false).

    When using both fixed resource domains and dynamic resource domains, you may allow adapter users to enter their own values, allowing them to add to the current list of values

for the resource domain. For more information, see "Registering Resource Domains" on page 74

- Enable the adapter check values to validate the adapter user-supplied values (set the setCanValidate method to true).

  As mentioned previously, to validate adapter user-supplied values you use callbacks known as adapter check values. The values are validated by the adapterCheckValue method. For more information, see "Adapter Check Value Callbacks" on page 80.

- Disable parameters in the Adapter Service Editor, by setting the setDisabled method to true.

For example, assume that a parameter named portNumber has a data type of int. When constructing the associated resource domain values, limit the values list to numeric strings as follows:

```
new ResourceDomainValues("portNumberLookup", new String[]
{"6048","8088","9090"});
```

The example above shows a ResourceDomainValues object with a set list. You have the option to define a range of values, by providing a minimum/maximum value. In this case, a single ResourceDomainValues object is used to define the minimum and maximum values of a numeric parameter. The conditions for doing this are as follows:

- The parameter must be of a numeric data type (such as int or long).
- The parameter cannot be a sequence parameter (array).
- The ResourceDomainValues object must contain exactly one value that is constructed from a ResourceDomainValue object.
- ResourceDomainValues.setComplete must be false. You must set this method explicitly if you used the ResourceDomainValue[] constructor; otherwise, changes to the parameter will not be saved.
- The embedded ResourceDomainValue object must have a name representing a numeric value, and an endName representing a number value greater that the value of the name.

  Note:
  This is the only defined use for ResourceDomainValue.endName. In all other cases, only ResourceDomainValue.name is used. All other ResourceDomainValue attributes are placeholders, and are not currently implemented. Using a String[] to construct ResourceDomainValues is equivalent to using a ResourceDomainValue[] where only the ResourceDomainValue.name attributes are populated.

Resource Domain Lookups

A dynamic resource domain uses resource domain lookups. That is, it uses addResourceDomainLookup to enable the adapter to look up parameter values in the adapter resource.

When adapter users create an adapter service node, they select a connection in which the adapter service executes. That connection also provides data from the resource domain that you registered with the connection.

When a node is created, the Adapter Service Editor initiates a resource domain lookup to the connection class for each of the adapter's metadata parameters that are associated with dynamic resource domains. Additional lookups are made as the adapter user modifies the values of parameters upon which other parameters depend (see"Associating Metadata Parameters with Resource Domains" on page 76).

All resource domain lookups invoke the same method in the WmManagedConnection implementation class:

```
ResourceDomainValues[] adapterResourceDomainLookup(String serviceName,
 String resourceDomainName,
 String[][] values)
```

where:

Argument	Description
serviceName	Always contains the class name of the adapter service.
resourceDomainName	Contains the registered name of the resource domain.
values	A multi-dimensional array that is populated with the current value of the parameters upon which the current lookup depends, as specified in the WmTemplateDescriptor.setResourceDomain call (as described in "Associating Metadata Parameters with Resource Domains" on page 76).

For example, consider the following call:

```
d.setResourceDomain( "columnsArg", "columnsLookup", new String[] {"tablesArg"});
```

This call creates a dependency on tablesArg (see "Associating Metadata Parameters with Resource Domains" on page 76). When the lookup for the columnsLookup resource domain is made, the values argument will contain the current settings for the tablesArg parameter. Data in the values argument is organized such that the first dimension of the array determines the dependent parameter, and the second dimension iterates the data in the parameter. Thus, values[0][0] contains the value of the first dependent parameter. If it is a sequence parameter, then values[0][1] would contain the next value in the sequence, values[0][2] the next, and so on. If there were more than one dependency, the contents of the second parameter on which the lookup depends would be contained in values[1][].

Note:
Placing a sequence parameter in a field map has several effects on the resource domain lookup process (see"Tuples" on page 82).

The signature of adapterResourceDomainLookup indicates that an array of ResourceDomainValues objects should be returned. Unless your implementation includes tuples, there should be exactly

one object in the response array, and the name attribute of that ResourceDomainValues object should always be the same as the resourceDomainName argument passed into the method.

When the Adapter Service Editor receives the lookup response data, it is evaluated against any data in the "current" parameter (the parameter for which the lookup was performed). An entry in the current parameter is considered valid if any one of the following is true:

- The parameter value matches a value in the ResourceDomainValues list
- ResourceDomainValues.setComplete(false) and ResourceDomainValues.setCanValidate(false)
- ResourceDomainValues.setComplete(false) and ResourceDomainValues.setCanValidate(true) and the value was successfully validated through the AdapterCheckValue facility

If the current parameter settings are considered valid, then the parameter values remain unchanged. However, any current parameter values are merged with the values in the lookup response when the parameter's drop-down list is opened. (This effectively extends the resource domain to include values that were entered by the user.) If the current parameter settings are not considered valid, then the invalid value will be deleted by a value in the resource domain.

Adapter Check Value Callbacks

When using both fixed resource domains and dynamic resource domains, you may allow adapter users to enter their own parameter values, allowing them to add to the current list of values for the resource domain. To enable the adapter to validate these values, you use callbacks known as *adapter check values*. Adapter check values are resource domain mechanisms that function very much like resource domain lookups. Any value in a parameter that is not part of the parameter's resource domain list may be validated by an adapter check value. For more information, see "Registering Resource Domains" on page 74.

Adapter check values can also validate resource domain lookups for each unique value that is not part of a resource domain list that is complete (that is, a list that does not accept adapter user-supplied values). For example, suppose the sequence parameter named colors contains the values "White", "Gray", "Black", and "Red". After a resource domain lookup, the resource domain list contains "Black" and "Gray". Assuming that the resource domain is configured appropriately, the Adapter Service Editor performs an adapter check value callback for "Red" and "White". If it finds either value, it deletes the cell containing that value (or overwrites it, if the sequence is in a field map).

To use an adapter check value callback, you must:

- Set the following methods of the ResourceDomainValues class as follows:
  - setComplete(false), which indicates that adapter users can enter their own parameter values; it sets a flag named complete
  - setCanValidate(true), which indicates that the adapterCheckValue method will validate user-supplied parameter values; it sets the canValidate flag

  If both of these flags are set properly, the Adapter Service Editor calls the following method for each value that is not already in the resource domain:

```
Boolean adapterCheckValue(String serviceName,
```

```
                        String resourceDomainName,
                        String[][] values,
                        String testValue)
```

The first three arguments are identical to those for adapterResourceDomainLookup (shown in "Resource Domain Lookups" on page 78). The testValue argument contains the value being checked.

- Register the adapter check value callback at the same time you register the resource domain name, using the WmAdapterAccess.addCheckValue method (see "Registering Resource Domains" on page 74).

Field Maps

As mentioned in "The WmDescriptor Interface" on page 60, you can use the createGroup method to organize parameters into different tabs, and to specify the order in which they appear. Similarly, you can use the createFieldMap method to organize various sequence parameters (within the same group) into a single table widget in the Adapter Service Editor.

The full signature for createFieldMap is:

```
void createFieldMap(String[] members,
                    boolean variable,
                    boolean fillAll);
```

where:

Argument	Description
members	A list of the parameter names that make up the field map. A field map may contain one or more member parameters, but a parameter should not be a member of more than one field map. The members argument is not an ordered list, and it has no impact on the order in which columns appear in a field map. The order of the columns in a field map is dictated entirely by the order in which the parameters appear in the group. The first member to appear in the group list will appear in the first column of the field map, and the remaining columns will follow in the order in which they appear in the group. If there is more than one field map in a group, then the relative positions of the first column parameters in the group will dictate the order in which the field map tables are displayed. Using parameters from different groups in a field map results in an exception.
variable	The value true permits adapter users to add rows to the table. In this case, the field map is considered to be a variable field map because the number of fields that appear in the Adapter Service Editor may vary. By default, each time an adapter user adds a row to the field map, each column is populated based on the associated parameter's data type and the contents of the associated resource domain. Typically, the column

Argument	Description
	will contain a drop-down list of string values from the resource domain. In other cases, either a check box appears (for Boolean parameters) or the column is empty.
fillAll	The value true fills the table with all available data. If the fillAll argument is true and the variable argument is false, then the table is expanded to contain one row for each value provided for the parameter in the first column of the field map. The values for this first column are provided by the associated resource domain, and cannot be changed or manipulated by the user. This is true even when the associated resource domain's setComplete method is false; the adapter user may not directly update this column. (The adapter user may still make changes to other parameters that might impact the content of the resource domain of the first column's parameter; see "Parameter Dependencies" on page 77.) The remaining columns will contain the same value and user interface widget that would be employed if the user manually inserted the row. If fillAll is false, then rows must be added by the user. **Note:** If variable is false, then fillAll is assumed to be true, regardless of the value passed in the argument.

Note:
Do not set both fillAll and variable to true; the resulting behavior might be unpredictable.

Tuples

The WmTemplateDescriptor.createTuple method is another grouping mechanism that you can use to modify:

- The behavior of a resource domain lookup.
- How resource domain values are applied to parameters in the tuple.

Members of a tuple are linked when the resource domain values are retrieved, and when the values are updated on the user interface. When the Adapter Service Editor performs a resource domain lookup for a parameter in a tuple, it expects the response array to contain a ResourceDomainValues object for each parameter in the tuple. Thus, changes resulting from the resource domain lookup will be applied simultaneously to each parameter in the tuple. This mechanism is particularly useful when two or more sequence parameters in a field map are closely related, for example when one parameter contains a column name and the other parameter contains the column format.

Requirements for reliable tuple operation are as follows:

- The parameters of a tuple must be sequence parameters of the same field map. If parameters are in separate field maps, the resource domain lookup will function properly, but the user interface characteristics described below will not function.

- A tuple must be declared (in the code) before a setResourceDomain method (see "Associating Metadata Parameters with Resource Domains" on page 76).
- The first parameter of a tuple must be assigned to a resource domain before any other member of the tuple (see "Associating Metadata Parameters with Resource Domains" on page 76).
- Each parameter of a tuple must have the same parameter dependencies listed in the setResourceDomain call that you used to assign the metadata parameters of the adapter service to a resource domain (as described in "Associating Metadata Parameters with Resource Domains" on page 76 and "Parameter Dependencies" on page 77).
- For each ResourceDomainValues object returned in the lookup, the setComplete method must be true (meaning that adapter users cannot supply parameter values; see "Populating Resource Domains with Values" on page 77).
- The first parameter in the tuple must appear first in its group.

In the user interface, the first parameter in a tuple serves as the "master" parameter, and all other parameters are "slave" parameters. In the Adapter Service Editor, the adapter user may directly manipulate the master parameter, but not slave parameters. A slave parameter contains the value from its resource domain that corresponds to the value selected from the master parameter. For example, if the fourth member of the master parameter's resource domain is selected, then the fourth member of the slave parameter's resource domain will appear in the slave parameter's column. If a slave parameter does not have a value in the position corresponding to the master parameter's value, then the slave parameter is left blank in that row.

Field Maps with Resource Domain Dependencies

As described in "Resource Domain Lookups" on page 78 and "Adapter Check Value Callbacks" on page 80, the addResourceDomainLookup and adapterCheckValue methods have a Values[][] argument that will contain the current value(s) of the parameter(s) on which the resource domain association depends. Normally, if the dependency parameter is a sequence parameter, the complete list of values is provided in the values argument. However, when a parameter depends on another parameter in the same field map, then by default each row in the field map is handled separately, for resource domain lookup/check value purposes.

For example, assume that with the sequence parameters A and B, the lookup for B depends on A. If the parameters are not in the same field map, then whenever a row in A changes, a lookup is performed for B, to which all values of A are passed. The results list is applied to each row of B, and updates are made to any rows of B that are no longer valid. (The rows may be no longer valid because their values are not members of a "complete" resource domain (that is, a resource domain that does not allow adapter users to enter values), or because an adapter check value callback failed to validate the rows.)

However, if A and B are in the same field map, then when the value in a row of A changes, only the new value in that row is passed to the resource domain lookup, and the values returned from the lookup apply only to that row of B. If parameter A contains catalog item numbers, for example, and parameter B contains the colors in which the item is available, then for each catalog item number in A, there would be a separate drop-down list of available colors in column B. Also note that if the same catalog item number appeared in multiple rows in column A, then the drop-down list of colors in column "B" would be the same. There would not be a separate resource domain

lookup for each of those rows because the adapter service editor would recognize that it already has the list of colors for that item number. Or, more accurately, that it already has a list of resource domain values based on the given set of dependency parameter values.

If you want to suppress the behavior described above, prefix the name of the parameter in the setResourceDomain dependency list with an asterisk (*). For example, the following method causes the server to treat parameter A like any other parameter, even if it were in a field map with B:

```
setResourceDomain("B", "bLookup", new String[] {"*A"})
```

Adapter Service Node Signatures

In addition to using the metadata model to create parameters for configuring an adapter service node, you can use the model to define the signatures of that node.

An adapter service node has an input signature and an output signature. An *input signature* describes the data that the service expects to find in the flow service pipeline at run time. An *output signature* describes the data that the service expects to add to the pipeline when it has successfully executed.

You can view an adapter service node's signature in the Input/Output tab of the Adapter Service Editor in Designer. Rules for defining signatures appear in the following procedure.

Once the signature is complete, users may include an adapter service node in flow constructs. They can route, map, and transform the input and output of the adapter service as needed in the integration solution.

The following procedure provides a basic model that you can use to implement a metadata signature. If you deviate from this model, it is important to understand that signature resource domains are only invoked as a result of a value applied to a dependent parameter. Having a resource domain lookup simply change the list of possible values in the resource domain will not impact the signature unless the current value is changed.

To create the signature of an adapter service node

1. Create metadata input parameters for the field names, data types, and signature. Each parameter must have a data type of String[].

   For example, assume that "inputNames", "inputTypes", and "inputSignature" are created as follows:

   ```
   public void setInputNames(String[] val);
   public void setInputTypes(String[] val);
   public void setInputSignature(String[] val);
   ```

2. Add these parameters to a group in the same order as above.

   For example:

   ```
   templateDescriptor.addGroup("group name", new String []
   { …, "inputNames", "inputTypes", "inputSignature"});
   ```

3. You may hide any or all of the parameters. (Hiding all parameters in a field map will hide the map table as well.)

   For example:

```
templateDescriptor.setHidden("inputNames");
templateDescriptor.setHidden("inputTypes");
templateDescriptor.setHidden("inputSignature");
```

4. Create a field map containing the three input parameters. In the createFieldMap method, set the variable argument to false, and set the fillAll argument to true.

   For example:

```
templateDescriptor.creatFieldMap(new String [] {"inputNames",
    "inputTypes", "inputSignature"}, false, true);
```

   In some cases, you can include other parameters in this field map, but this can sometimes be problematic, particularly if these parameters are hidden.

5. Create a tuple containing the names and types parameters.

   For example:

```
templateDescriptor.creatTuple(new String [] {"inputNames",
    "inputTypes"});
```

6. In the associated connection class(es), register two resource domains to support name and type lookups.

   For example, assume that the resource domains "inputNamesLookup" and "inputTypesLookup" are created as follows.

```
access.addResourceDomainLookup("inputNamesLookup", this);
access.addResourceDomainLookup("inputTypesLookup", this);
```

7. Assign the names parameter to the name lookup resource domain, and assign the types parameter to the type lookup resource domain. These assignments must specify the same dependencies because the parameters are in a tuple. If no dependencies are known at this time, specify null.

   For example:

```
templateDescriptor.setResourceDomain("inputNames","inputNamesLookup",null);
templateDescriptor.setResourceDomain("inputTypes","inputTypesLookup",null);
```

8. Assign the signature parameter to one of the reserved resource domain names provided in WmTemplateDescriptor, specifying the names parameter and the types parameter as dependencies.

   For example, "INPUT_FIELD_NAMES" would be used as follows:

```
templateDescriptor.setResourcDomain( "inputSignature",
    WmTemplateDescriptor.INPUT_FIELD_NAMES, new String[]
```

```
    {"inputNames", "inputTypes"}););
```

9. Implement the name and type lookups as described in "Implementing Resource Domain Lookups for Signature Names and Data Types" on page 86.

10. Create an output signature by repeating this procedure, substituting OUTPUT_FIELD_NAMES for INPUT_FIELD_NAMES in step 8.

Implementing Resource Domain Lookups for Signature Names and Data Types

As mentioned in "Adapter Service Node Signatures" on page 84, typically you load a signature's name and data type parameters using resource domain lookups. You implement lookups by including an adapterResourceDomainLookup method in your WmManagedConnection implementation class.

These parameters are implemented as string arrays, with the corresponding index in each array used to associate a name with a data type. The following subsections describe the values you supply in your resource domains for the name and data type parameters.

Field Name String Values

A field name string can contain a simple value, such as "itemNumber", or a more complex value such as "customer.orders[].lineItems[].itemNumber". The complex field name string demonstrates the ability of a signature to specify hierarchy (using a dot ".") and multiplicity (using a pair a square brackets "[]"). Thus, this example shows an aggregate of customer fields containing multiple orders that may contain multiple line items that contain one itemNumber.

Follow these rules when creating resource domain values for containing signature names:

- A name can be used as a field (containing data) or an aggregate (containing fields), but not both.
- Each entry should contain a field with any containing aggregates. Do not specify aggregates alone.
- Both fields and aggregates can be arrays, indicated by square brackets. For example:

  "customer.orders[].lineItems[].itemNumber"

- Use name restrictions. For more information, see the *webMethods Service Development Help* for your release. Because they are less strict than Java variable name constraints, there should never be a problem.

Data Type String Values

A data type string should contain the data type corresponding to the field name string at the same index in the field name's resource domain values list. Data types for adapter services are similar to data types for Java services (for more information, see the *webMethods Service Development Help* for your release). That is, if the signature item is to be accessible from an Integration Server flow, its data type should be "java.lang.String", "java.util.Date", or one of the "big-letter-primitive" classes

(e.g., java.lang.Integer). If the data need not be accessible from a flow, then any class type is acceptable.

Multiplicity in the data type string uses a pair of square brackets [] appended to the class name. Data type multiplicity represents the multiplicity across the entire signature hierarchy by adding a set of brackets for each set of brackets in the corresponding name string.

Even though there is only one item number in the lineItems aggregate, there are many in the signature. In this case, the data type would be "java.lang.Integer[][]" (assuming itemNumber is an integer).

For examples, see "Code Example 1: WmAdapterService Implementation Class" on page 99. For more information, see "Interacting with the Pipeline" on page 87.

Adapter Service Execution

When a flow service, trigger, Designer invokes an adapter service node, the service calls the WmAdapterService.execute method. This method receives a WmManagedConnection object and a WmRecord object, and it is expected to return a WmRecord object.

The connection object argument delivers a connection instance from the adapter service's connection node. How the adapter uses this connection object to gain access to the adapter resource is determined by the adapter's design, not by the ADK.

The inbound WmRecord object will contain data based on the input signature as well as any other information that may be in the flow service pipeline at the time the adapter service is invoked. The execute method is responsible for interrogating the inbound WmRecord object to retrieve data necessary for the adapter service to perform its function. The server will not validate that the fields specified in the signature are actually present in the pipeline, but the data type for any field present in the pipeline is guaranteed to conform to the data type specified in the signature. The adapter service must determine whether all required data is present, and how to respond when data elements are missing.

When the execute method completes, the outbound WmRecord object should contain data based on the output signature defined in the metadata. This data will be added to the pipeline. (An adapter service may not remove data from the pipeline because it only works with a copy of the pipeline object, not the original pipeline object.) Once again, the adapter implementation has primary responsibility for determining what portions of the signature will be populated. A run time exception might occur if a field that was not populated by the adapter service is mapped within a flow service.

Interacting with the Pipeline

At run time, an adapter service needs to retrieve data from the pipeline at the beginning of its execution, and to add data to the pipeline at the end of execution.

To retrieve data from the pipeline, the service should interrogate the WmRecord argument of the execute method. Limit the interrogation to fields identified in the service's signature because often there is other information in the pipeline that is not intended for the service.

At the end of execution, the execute method should return a WmRecord instance containing data that should be added to the pipeline. Organize the return data in a way that is consistent with the metadata signature so that other adapter services (or flow or Java services) can access it.

For discussion purposes, assume the following as a sample metadata signature for both input and output of an adapter service:

Field Name	Type
customer.id	java.lang.Integer
customer.name	java.lang.String
customer.orders[].id	java.lang.Integer[]
customer.orders[].date	java.util.Date[]
customer.orders[].lineItems[].itemNumber	java.lang.Integer[][]
customer.orders[].lineItems[].quantity	java.lang.Integer[][]
customer.orders[].lineItems[].description	java.lang.String [][]

The sample signature describes a hierarchal structure that can be expressed as a tree structure, where the actual field names form the leaves, and the elements preceding the field name are nodes. Thus, the names customer, orders, and lineItems are node names, and id, name, date, itemNumber, quantity, and description are leaves in the tree structure.

The WmRecord class, which is the primary carrier of data into and out of adapter services, is a JCA-based wrapper for an IData object. It provides methods that access the IData content. However, when dealing with a hierarchal structure (as is the case with the sample), it is necessary to "drill down" into the IData structure. Therefore, ignore the WmRecord methods except for getIData, and putIData (which is used to access the underlying IData object).

Note:
The IData interface is part of the standard webMethods Integration Server Java API. Its structure is based on key/value pairs, where the key is a String and the value is a Java object. For more information, see the Javadoc entries for IData, IDataCursor, IDataFactory, and IDataUtil, located in the *Integration Server_directory* \doc\api\Java directory.

The getIData Method

The WmRecord.getIData method returns an IData object that contains the entire pipeline at the time the service was invoked. The top-level branch or leaf name(s) in the metadata signature (in this case, customer) is the key used to access data intended for use by the service. The value associated with that key will be either another IData object (if the key is a node name) or an object of the type specified by the corresponding type field of the signature. If the name of the branch or leaf includes a pair of square braces "[]", then the value will contain an array of the designated object type.

Thus, the fields in the sample would be populated as follows:

- WmRecord.getIData would return an IData object with an entry, keyed with the name customer.
- The value associated with customer would be another IData object, this time with three entries: id, name, and orders.
- The value corresponding with id would be an Integer.
- The value of name would be a String.
- The value of orders would contain an array of IData objects.
- The orders IData objects would each contain an Integer id, a java.util.Date date, and an array of IData objects associated with the lineItems key.
- The lineItems IData objects would contain entries for itemNumber, quantity, and description, with the data types provided in the signature.

When constructing response data to place in the pipeline at the end of service execution, use the same rules that apply to interpreting the metadata signature. For each node level in the signature, there should be a corresponding layer of IData, keyed with the names from the signature. For each leaf, there should be an IData entry with the corresponding signature name and type.

Note:
Signatures are not enforced by Integration Server or the ADK framework. The validity of a request based on the presence or absence of a given field, or the value given to a field, is determined exclusively by the adapter implementation. Similarly, if the service fails to populate the response WmRecord with data organized according to the signature, subsequent services will be unable to access the data provided by the adapter service.

Adapter Service Template Interactions

Creating Adapter Service Nodes

Designer provides a wizard to guide the adapter user through the process of creating adapter service nodes. During this process there are two significant interactions with the adapter. They occur:

- When the adapter user selects the connection node to be used by the service node.
- After the adapter user enters the name and folder of the new adapter service node.

Selecting Connection Nodes

When the adapter user selects the connection node to be used by the service node, the server calls the fillResourceAdapterMetadataInfo method in the connection for the supported adapter service template class names. Then the server:

- Instantiates each of these template classes.
- Retrieves the default metadata parameter values (by calling the parameter "get" methods).

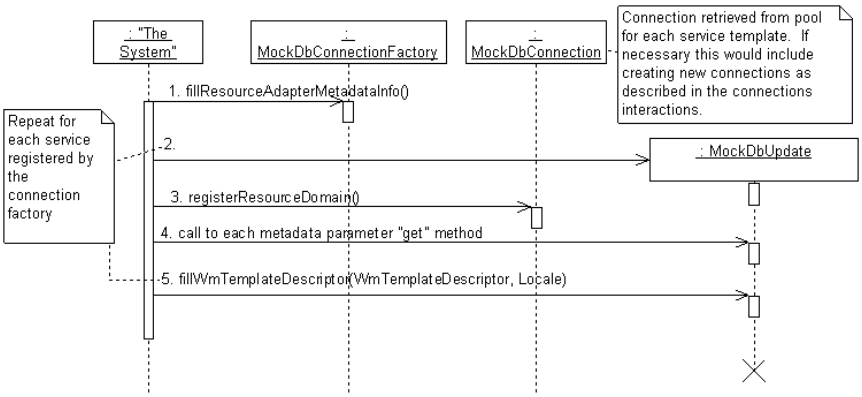
- Calls its fillWmTemplateDescriptor method.

All this information is cached in Designer session, and will not be requested again for any adapter service activity associated with that connection node. That is, after this information has been gathered from a connection node, the adapter user may create multiple adapter service nodes based on any template associated with that connection using the same set of cached information. This is particularly significant during development of metadata-related code.

Note:
The cache is not cleared when you recompile the code or reload the package, so it is critical that you refresh the cache manually when loading updated metadata code. (To refresh the cache, use the Refresh button on the Designer toolbar or select Refresh from the Session menu.)

The following diagram shows the adapter calls made when adapter users select a connection node.

Loading metadata cache in Designer



Entering Names and Folders for Adapter Service Nodes

The other significant adapter interaction that occurs when creating adapter service nodes occurs after the adapter user has entered the name and folder of the new adapter service node.

Before displaying the Adapter Service Editor screens, the following occurs:

- The server invokes the connection's adapterResourceDomainLookup method for each lookup registered with the service.

  The values argument in these lookups will reflect the dependent parameters' default values that are cached when the adapter user selects a connection node. If your default value for a dependent parameter is null, make sure your resource domain lookup code can handle a null value in the values argument.

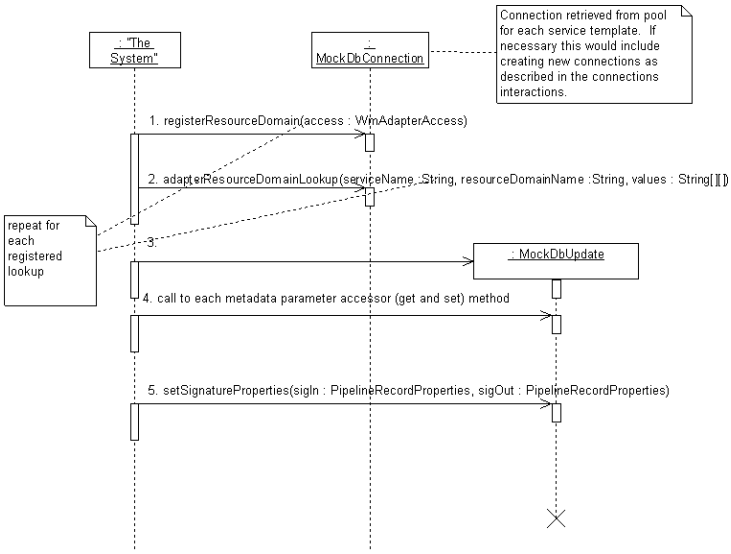
- After the lookups are complete, the server instantiates the adapter service template class (again), and each of the accessor methods are called.

  Values passed to "set" methods come either from the parameter default, or from the result of a resource domain lookup. These accessor method calls merely validate their operation; the service class instance is not cached. This is the last interaction with the metadata parameter

accessor methods during the process of creating an adapter service node. The "set" methods are not called with the final node settings until the service is executed.

The following figure describes this interaction.

Parameter interrogation during the creation of an adapter service node



Viewing or Editing Adapter Service Nodes

Users may modify the metadata parameter values of an adapter service node using the Adapter Service Editor at any time after the node is created.

When an adapter service node is opened (selected), the Adapter Service Editor performs an adapterResourceDomainLookup call for any resource domain values that it has not already cached. (This lookup interaction is shown in steps 1 and 2 of the above figure.)

Resource domain values are cached in the Adapter Service Editor based on the values of dependent parameters for the adapter service template/connection type combination (that is, for the class, not the node; thus cached values may be used across nodes that are based on the same template and connection type). Whenever an adapter user changes the value of a dependency parameter (a parameter upon which a resource domain lookup depends), the Adapter Service Editor checks its cache for a set of resource domain values based on the new value. If an appropriate set of resource domain values is not found in the cache, then the Adapter Service Editor will call the adapterResourceDomainLookup method again.

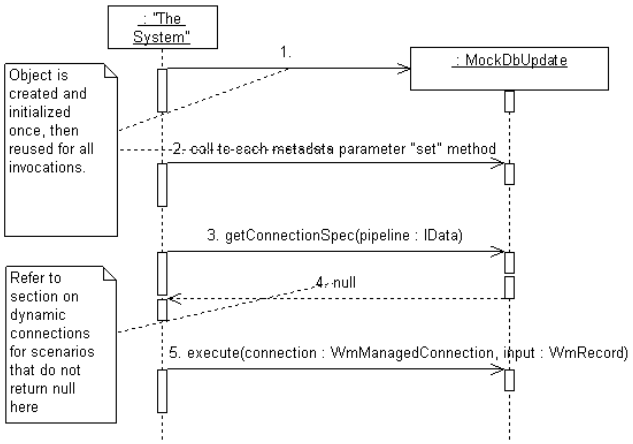
Adapter Check Values operate very much like resource domain lookups. When an adapter user types a value into a parameter configured with an adapter check value, a call is made to the adapterCheckValue method of the corresponding connection class. If the validation succeeds, the Adapter Service Editor caches the checked parameter value as well as the values of any dependent parameters for future use. If a parameter uses the adapter check value feature and is also a dependency parameter for the resource domain lookup of another parameter, the adapter check

value validation is performed first. If the validation succeeds, the appropriate lookups are performed. For more information, see "Adapter Check Value Callbacks" on page 80

Executing Adapter Service Nodes

When an adapter service executes, the server first creates a new instance of the corresponding class, unless it is already cached. Then it calls the metadata parameter "set" methods, passing the parameter settings stored in the adapter service node. Lastly, it calls the service's execute method, passing a connection instance and a copy of the pipeline wrapped in a WmRecord object, as shown in the following figure.

Adapter service execution



Adapter Service Implementation

The example provided in this section demonstrates the mechanics of an adapter service implementation by making full use of design-time and run-time interactions while emulating interactions with an external adapter resource.

The example adapter service simulates a simple database update service, allowing an adapter user to select table and column names that create the run-time signature for the resulting adapter service node. Table and column names are provided from hard-coded lists in a mock-connection implementation, as if the data were actually retrieved from the adapter resource. The example also includes interactions with the pipeline based on a dynamic service signature. The example is self-contained; it does not actually interact with any external resource.

The model for adapter services forces syntactic and semantic coupling of code in different methods and classes. Because of this, it might be difficult to understand the process of creating an adapter service by looking at classes (or even methods) as a unit of work in the development process. For example, adding a metadata parameter can require updating two or more methods in the Adapter Service implementation class, updating up to three methods in the associated connection classes, and adding two entries in the resource bundle. Thus, this section approaches the implementation of a service as a series of activities that you perform, each of which may traverse multiple methods and classes.

This section provides examples of the resulting code, and refers to specific lines.

The tasks for creating an adapter service are as follows:

- "Defining a WmAdapterService Implementation Class" on page 93.
- "Updating the Resource Bundle" on page 93.
- "Specifying Adapter Service Configuration Data" on page 93.
- "Implementing Adapter Service Configuration Resource Domains" on page 95.
- "Manipulating Adapter Service Signature Properties" on page 96
- "Specifying Adapter Service Signature Data" on page 98.
- "Specifying Adapter Service Signature Resource Domains" on page 99.
- "Implementing the WmAdapterService.execute Method" on page 99.

Defining a WmAdapterService Implementation Class

Create a class that extends com.wm.adk.cci.interaction.WmAdapterService (as shown in line 20 of "Code Example 1: WmAdapterService Implementation Class" on page 99). In order to compile, this class must implement the abstract method named execute (line 112). In addition, you must override the base class implementation of the fillWmTemplateDescriptor method (line 59).

Updating the Resource Bundle

Optionally, update the resource bundle with a display name, description, and help URL to make the service more usable, as shown in lines 1-6 of "Code Example 3: Resource Bundle Updates" on page 105.

Specifying Adapter Service Configuration Data

The next logical step for implementing an adapter service is to create the metadata that enables the adapter user to create adapter service nodes. To do this, you create metadata parameters appropriate for the function of the adapter service, and describe presentation and data entry rules for those parameters.

Creating "Data Entry" Parameters

The sample implementation includes five metadata parameters that adapter users will use for data entry when they create adapter service nodes. Each parameter has:

- An accessor method (as shown in lines 53-57 of "Code Example 1: WmAdapterService Implementation Class" on page 99).
- A variable to hold the configured values (lines 42-46 of "Code Example 1: WmAdapterService Implementation Class" on page 99).
- A String constant containing the name of the parameter (lines 27-31 of "Code Example 1: WmAdapterService Implementation Class" on page 99).

- A set of resource bundle entries with a localizable parameter name and description (lines 13-36 of "Code Example 3: Resource Bundle Updates" on page 105). (For information about metadata parameters, see "webMethods Metadata Parameters" on page 58.)

The following table describes the purpose of each of these "data entry" parameters:

Parameter	Description
tableName	Enables adapter users to select a table to update.
columnNames	Lists the columns in the selected table for updating. The resource domain lookup for this parameter will depend on the value in the tableName parameter.
columnTypes	Contains the default type associated with the columnName.
overrideTypes	Enables adapter users to type a different data type for the column value.
repeating	A Boolean flag indicating whether the service will be used to update a single row or multiple rows of the table.

Specifying the Display and Data Entry Attributes of the "Data Entry" Parameters

After you create the parameters, you specify their display and data entry attributes by calling various methods of the WmTemplateDescriptor interface from the service's fillWmTemplateDescriptor method. The example code places each data entry parameter into a single group (in display order) referenced by the constant UPD_SETTINGS_GRP. (A constant is used to name the group (instead of a string) because the same value is used in the resource bundle to specify a localizable group name and help URL; see lines 8-11 of "Code Example 3: Resource Bundle Updates" on page 105.)

Placing the Column Names and Column Data Types Parameters in a Field Map

Next, the example places columnNames and the two types parameters in a fieldMap (lines 73-75). The use of two data type columns in the fieldMap warrants further discussion. The desired behavior is to automatically update the data type when the column name changes, but to also allow the adapter user to enter an alternative data type for the given field and to (presumably) have the adapter convert the data at run time. In order to have the type value change with the columnName, the resource domain associated with the parameter must be "complete" (as specified by the setComplete method being set to true; see "Registering Resource Domains" on page 74). Users are not allowed to type values into fields of a "complete" resource domain. Because of these conflicting constraints, it is necessary to have two parameters with different resource domain associations: one updated by the Adapter Service Editor, and the other by the adapter user.

Placing the Column Names and Column Types Parameters in a Tuple

Finally, the columnNames and columnTypes parameters are placed in a tuple (line 77). In a tuple, not only are the resource domain lookups performed together, but the Adapter Service Editor maintains a relationship between the parameter settings such that when value[n] is selected from the resource domain value list for columnNames, then value[n] is automatically selected for the

corresponding columnTypes value. (Alternatively, you can make a resource domain lookup for columnTypes that depends on the corresponding value of columnNames and then determines the appropriate type in the lookup.)

Implementing Adapter Service Configuration Resource Domains

The next step for implementing an adapter service is to:

- Define and implement the resource domains required for the parameters that you established in "Specifying Adapter Service Configuration Data" on page 93.
- Identify the values upon which those resource domains depend.

For each parameter that requires either a resource domain to supply a value or the validation of user-supplied values, you must:

- Call WmTemplateDescriptor.setResourceDomain, passing the name of the parameter, the name of the resource domain, and an array of the names of any parameters on which the resource domain will depend (see lines 86-102 of "Code Example 1: WmAdapterService Implementation Class" on page 99).
- Register the resource domain support in the associated connection class's registerResourceDomain method (see lines 104-117 of "Code Example 2: WmManagedConnection Implementation Class Updates" on page 103).
- Implement code to populate the resource domain values and/or the "check values". This is discussed below.

The first parameter, tableName, is associated with a new resource domain called tableNameRD (line 86 of "Code Example 1: WmAdapterService Implementation Class" on page 99). The list of available tables in this example does not depend on any other parameters, so the dependency list on line 86 is left null. Furthermore, since the list does not change once it is retrieved from the resource, it can be implemented as a complete resource domain during registration (lines 127-130 of "Code Example 2: WmManagedConnection Implementation Class Updates" on page 103). When a ResourceDomainValues object is provided in registerResourceDomain, there is no need to add support for the resource domain in the lookup method.

The resource domains columnNamesRD and columnTypesRD are created to support the columnNames and columnTypes parameters, respectively. The columNamesRD resource domain depends on the value of the tableName parameter, and since columnNames and columnTypes were placed in a tuple, columnTypesRD must also depend on the value of tableName. The lookup implementation for these resource domains (lines 39-57 of "Code Example 2: WmManagedConnection Implementation Class Updates" on page 103) checks for the name of either resource domain and it returns the values for both resource domains in a single response array. (The columnNamesRD resource domain will always be passed as the resource domain name in this lookup. Adding the '||' check on line 40 decouples the lookup implementation from the order in which the parameters were placed in the tuple.)

The final resource domain in this section, OverrideTypesRD (associated with the "overrideTypes" parameter on line 91 of "Code Example 1: WmAdapterService Implementation Class" on page 99), is used to validate data entered by adapter users into a Java class that will contain the data at run

time. The specifics of the check implementation (lines 16-32 of "Code Example 2: WmManagedConnection Implementation Class Updates" on page 103) are probably unrealistic for a real-world environment, but all the mechanics of doing a check are demonstrated. Remember to set the setCanValidate method to true in the resource domain (line 115 of "Code Example 2: WmManagedConnection Implementation Class Updates" on page 103).

Manipulating Adapter Service Signature Properties

In addition to controlling the names and types of the fields in an adapter service signature, the adapter has control over several other aspects of a service's structure and behavior. These aspects can be broadly divided into two categories: Template-based properties which apply to all adapter service nodes associated with the service implementation class; and signature field properties which are specific to the records and fields that make up the signature of a particular node.

Template-Based Signature Properties

In the implementation of the adapter service's fillWmTemplateDescriptor() method, the following signature-related features may be configured using the following methods:

- **WmDescriptor.setShowConnectionName(boolean)** - When set to `true`, the reserved string field named "$connectionName" is included in the service input signature. This field allows flow writers to control the connection pool used during each invocation of the service. The "$connectionName" field is inserted in the top-level signature record, outside the wrapper. See the javadoc for WmDescriptor.showConnectionName() for more information on the effect of "$connectionName" on the run time behavior of the service. (Default = `True`.)
- **WmDescriptor.setSignatureWrapped(boolean)** - When set to `true`, adapter-defined signature fields are wrapped in a record called xxxInput for input fields and xxxOutput for output fields, where xxx represents the name of the adapter service node. (Default = `True`).
- **WmDescriptor.setPassFullPipeline(boolean)** - When set to true, the adapter service has access to all fields in the pipeline; when set to false, the adapter service may only read or write to pipeline fields that are part of the service's defined signature. (Default = `False`.)

Signature Field Properties

All Integration Server services (including adapter services, flow services, and java services) contain an input and output signature definition that identifies the names and types of the pipeline fields read or written by the service. For each of these fields there are also a number of properties that can be used to document the intended use of the field or to create constraints on what data is valid for that field at run time. For more information on these properties, see the *webMethods Service Development Help* for your release.

In flow services and java services, signature fields and their properties can be modified by the user. For adapter services, the adapter defines resource domains that act as callbacks from the developer tool that allow the adapter to specify the name, data type, and structure of the service's signature while the service is being configured. A separate callback mechanism allows the adapter to control a limited subset of signature field constraint properties.

The signature field constraints that are controlled by the adapter include:

- **Required** - If true, the field must be present on the pipeline.
- **Allow null** - If false, when the field is present, it cannot hold a null value.
- **Allow unspecified fields** - Controls whether the document (record) element can hold fields that are not specifically identified in the signature. (This constraint has no effect on non-document-type signature elements.)

The controls for these constraints are disabled so the user can view the value, but not modify it. The remaining properties (that are not related to name and type) are enabled and may be managed like any other service.

Note:
Like other service types, signature constraints are only enforced at run time if the associated "Validate input" or "Validate output" check box is selected by the user.

Adapters gain access to a signature field's constraints by overriding the setSignatureProperties() method in the WmAdapterService implementation class. This method is called whenever the adapter service node is saved, and whenever the user views the input/output panel of an unsaved adapter service. Because this call is made while adapter service changes are still in progress, the call is made against a temporary object that has all metadata parameter settings as they currently exist in the adapter service editor. This is not the same object that is used for run-time service invocations.

The ADK includes three classes that are used to provide access to signature property information.

- **PipelineVariableProperties** is an abstract base class that represents any signature element (field or record). It exposes read access to all common signature element properties (name, data type, comments, etc.) and read/write access to the "Required" and "Allow null" constraints.
- **PipelineFieldProperties** is a concrete subclass of PipelineVariableProperties that represents a single non-document-type field in the signature. It adds read-only access to properties that are specific to non-document elements (pick list choices, content type, etc.)
- **PipelineRecordProperties** is a concrete subclass of PipelineVariableProperties that represents a single document-type element in the signature. It adds read/write access to the "Allow unspecified fields" property, and methods for accessing the member elements of the record.

The WmAdapterService.setSignatureProperties() method receives two PipelineRecordProperties objects as arguments, one for the input signature and the other for the output signature. The adapter should use the accessor/navigation methods available through that object to locate signature elements for which constraints need to be managed. Within the setSignatureProperties() implementation, it is frequently useful to use the WmAdapterService.inputRecordName() and outputRecordName() methods to get the names of the respective signature wrappers (if wrappers are enabled). If a connection is needed to get constraint information from a connection target, use the WmAdapterService.retrieveConnection() method.

Note:
This is the only place retrieveConnection should be used; using it from the service's execute method will cause errors.

The code listing below demonstrates a simple implementation of signature constraint management.

```
    protected void setSignatureProperties(
            PipelineRecordProperties inputSigProps,
            PipelineRecordProperties outputSigProps) throws ResourceException
    {
        if(this._tableName != null)
        {
           // need a connection to look up info about table being updated
           MockDbConnection conn = (MockDbConnection)retrieveConnection();
           TableInfo info = conn.getTableInfo(this._tableName);
           updateSignatureRecord(inputSigProps, info, _columnNames,
                   inputRecordName());
           updateSignatureRecord(outputSigProps, info, _columnNames,
                   outputRecordName());
        }
    }
    private void updateSignatureRecord(PipelineRecordProperties inputSigProps,
            TableInfo tInfo, String[] columnNames, String wrapperName)
    {
        PipelineRecordProperties wrapperRec =
       (PipelineRecordProperties)inputSigProps.findByPath(wrapperName);
   wrapperRec.setAllowNull(false);
   wrapperRec.setAllowUnspecifiedFields(false);
   wrapperRec.setRequired(true);
   if(columnNames != null)
   {
       for(int i = 0; i < columnNames.length; i++)
       {
           ColumnInfo cInfo = tInfo.getColumnInfo(columnNames[i]);
           PipelineVariableProperties fieldProps =
               wrapperRec.findByPath(columnNames[i]);
           fieldProps.setAllowNull(false);
           fieldProps.setRequired(cInfo.isRequired());
       }
   }
}
```

Specifying Adapter Service Signature Data

After you have implemented the configuration logic for the adapter service template, you implement logic that defines the run-time signature of a configured adapter service node. To begin doing this, you create the following additional metadata parameters:

- sigIn and sigOut

  These parameters are for use with the reserved signature resource domains.

- fieldNames and fieldTypes

  These parameters are the dependency parameters in which you build the signature data. Because the example update service has the same input and output signature, only one set of name and type parameters is necessary. The relationship between these parameters is established in the WmTemplateDescriptor on lines 84-87 of "Code Example 1: WmAdapterService Implementation Class" on page 99. The mechanics of signature construction was discussed in "Adapter Service Node Signatures" on page 84.

These parameters will not accept adapter user input; from the adapter user's perspective, they are largely redundant with information provided elsewhere in the Adapter Service Editor. In most implementations, they would be included in the same group with the configuration parameters, but would be hidden from the adapter user. For demonstration purposes, these parameters remain visible in the example, but they are located in a separate group. Except for that, the metadata constructs for these parameters follow the same basic rules for specifying a signature for any service.

Specifying Adapter Service Signature Resource Domains

The resource domain implementation for the signature parameters is a little more complex. The fieldNamesRD and fieldTypesRD resource domains are designed to depend on the values of each of the configuration parameters from "Specifying Adapter Service Configuration Data" on page 93. In the lookup implementation (lines 58-99 of "Code Example 2: WmManagedConnection Implementation Class Updates" on page 103), both fieldNamesRD and fieldTypesRD are constructed as "complete" resource domains.

For fieldNamesRD, the tableName and columnName parameter values are used to form a hierarchical signature in which the columnName elements are contained in an aggregate named by the tableName. If the parameter named repeating is set to true, then the tableName aggregate is converted to an array by inserting square brackets ("[]") in the field name.

For fieldTypesRD, a value in the overrideTypes parameter has precedence over a value in columnTypes, and the repeating parameter is used to specify whether the type repeats.

Implementing the WmAdapterService.execute Method

The final step for implementing an adapter service is to implement its execute method. Typically, most of the logic of this method is specific to the resource with which the adapter communicates. In nearly all cases the adapter must interact with the pipeline at the beginning and/or end of the execute method. The methods unpackRequest and packResponse (lines 119 and 176, respectively, of "Code Example 1: WmAdapterService Implementation Class" on page 99) demonstrate an effective method of interacting with the pipeline using the same metadata parameters that were used to create the signature.

Important:
The unpackRequest and packResponse methods read class fields, but they never write to them. This is important because of the multi-threaded nature of adapter service execution. At run time, exactly one WmAdapterService object corresponds to each adapter service node defined in the namespace. All invocations of a given adapter service node call the execute method on the same object. If more than one thread is executing the service at the same time, then updates to class fields by one thread will inevitably collide with those of another thread.

Code Example 1: WmAdapterService Implementation Class

```
1.   package com.mycompany.adapter.myadapter.services;
2.
3.   import java.util.Hashtable;
4.   import java.util.Locale;
```

```
5.
6.   import javax.resource.ResourceException;
7.
8.   import com.mycompany.adapter.myadapter.MyAdapter;
9.   import com.wm.adk.cci.interaction.WmAdapterService;
10.  import com.wm.adk.cci.record.WmRecord;
11.  import com.wm.adk.cci.record.WmRecordFactory;
12.  import com.wm.adk.connection.WmManagedConnection;
13.  import com.wm.adk.metadata.WmTemplateDescriptor;
14.  import com.wm.data.IData;
15.  import com.wm.data.IDataCursor;
16.  import com.wm.data.IDataFactory;
17.  import com.wm.data.IDataUtil;
18.
19.
20.  public class MockDbUpdate extends WmAdapterService
21.  {
22.
23.     public static final String SIG_IN_PARM = "sigIn";
24.     public static final String SIG_OUT_PARM = "sigOut";
25.     public static final String FIELD_NAMES_PARM = "fieldNames";
26.     public static final String FIELD_TYPES_PARM = "fieldTypes";
27.     public static final String TABLE_NAME_PARM = "tableName";
28.     public static final String COLUMN_NAMES_PARM = "columnNames";
29.     public static final String COLUMN_TYPES_PARM = "columnTypes";
30.     public static final String REPEATING_PARM = "repeating";
31.     public static final String OVERRIDE_TYPES_PARM = "overrideTypes";
32.
33.     public static final String TABLES_RD = "tablesRD";
34.     public static final String COLUMN_NAMES_RD = "columnNamesRD";
35.     public static final String COLUMN_TYPES_RD = "columnTypesRD";
36.     public static final String FIELD_NAMES_RD = "fieldNamesRD";
37.     public static final String FIELD_TYPES_RD = "fieldTypesRD";
38.     public static final String OVERRIDE_TYPES_RD = "overrideTypesRD";
39.
40.     private String[] _fieldNames;
41.     private String[] _fieldTypes;
42.     private String _tableName;
43.     private String[] _columnNames;
44.     private String[] _columnTypes;
45.     private boolean _repeating;
46.     private String[] _overrideTypes;
47.
48.
49.     public void setSigIn(String[] val){}
50.     public void setSigOut(String[] val){}
51.     public void setFieldNames(String[] val){ _fieldNames = val;}
52.     public void setFieldTypes(String[] val){ _fieldTypes = val;}
53.     public void setTableName(String val){ _tableName = val;}
54.     public void setColumnNames(String[] val){ _columnNames = val;}
55.     public void setColumnTypes(String[] val){ _columnTypes = val;}
56.     public void setRepeating(boolean val){ _repeating = val;}
57.     public void setOverrideTypes(String[] val){_overrideTypes = val;}
58.
59.     public void fillWmTemplateDescriptor(WmTemplateDescriptor d,Locale l)
60.        throws ResourceException
61.     {
62.        d.createGroup("Mock Settings", new String [] { TABLE_NAME_PARM,
63.                                                       REPEATING_PARM,
64.                                                       COLUMN_NAMES_PARM,
```

```
65.                                                  COLUMN_TYPES_PARM,
66.                                                  OVERRIDE_TYPES_PARM});
67.
68.        d.createGroup("Signature", new String [] {FIELD_NAMES_PARM,
69.                                                  FIELD_TYPES_PARM,
70.                                                  SIG_IN_PARM,
71.                                                  SIG_OUT_PARM});
72.
73.        d.createFieldMap(new String[] {COLUMN_NAMES_PARM,
74.                                       COLUMN_TYPES_PARM,
75.                                       OVERRIDE_TYPES_PARM},true);
76.
77.        d.createTuple(new String[]{COLUMN_NAMES_PARM,COLUMN_TYPES_PARM});
78.
79.        d.createFieldMap(new String [] {FIELD_NAMES_PARM,
80.                                        FIELD_TYPES_PARM,
81.                                        SIG_IN_PARM,
82.                                        SIG_OUT_PARM},false);
83.
84.        d.createTuple(new String[]{FIELD_NAMES_PARM,FIELD_TYPES_PARM});
85.
86.        d.setResourceDomain(TABLE_NAME_PARM,TABLES_RD,null);
87.        d.setResourceDomain(COLUMN_NAMES_PARM,COLUMN_NAMES_RD,
88.           new String[]{TABLE_NAME_PARM});
89.        d.setResourceDomain(COLUMN_TYPES_PARM,COLUMN_TYPES_RD,
90.           new String[]{TABLE_NAME_PARM});
91.        d.setResourceDomain(OVERRIDE_TYPES_PARM,OVERRIDE_TYPES_RD,null);
92.
93.        String [] fieldTupleDependencies = {TABLE_NAME_PARM,
94.                                            REPEATING_PARM,
95.                                            COLUMN_NAMES_PARM,
96.                                            COLUMN_TYPES_PARM,
97.                                            OVERRIDE_TYPES_PARM};
98.
99.        d.setResourceDomain(FIELD_NAMES_PARM,FIELD_NAMES_RD,
100.          fieldTupleDependencies);
101.       d.setResourceDomain(FIELD_TYPES_PARM,FIELD_TYPES_RD,
102.          fieldTupleDependencies);
103.
104.       d.setResourceDomain(SIG_IN_PARM,WmTemplateDescriptor.INPUT_FIELD_
NAMES,
105.          new String[] {FIELD_NAMES_PARM, FIELD_TYPES_PARM});
106.       d.setResourceDomain(SIG_OUT_PARM,WmTemplateDescriptor.OUTPUT_FIEL
D_NAMES,
107.          new String[] {FIELD_NAMES_PARM, FIELD_TYPES_PARM});
108.
109.       d.setDescriptions(MyAdapter.getInstance().
110.          getAdapterResourceBundleManager(),l);
111. }
112.    public WmRecord execute(WmManagedConnection connection, WmRecord input)
113.       throws ResourceException
114.    {
115.       Hashtable[] request = this.unpackRequest(input);
116.       return this.packResonse(request);
117.    }
118.
119.    private Hashtable[] unpackRequest(WmRecord request) throws
ResourceException
120.    {
121.       Hashtable data[] = null;
```

```
122.       IData mainIData = request.getIData();
123.       IDataCursor mainCursor = mainIData.getCursor();
124.
125.       try
126.       {
127.          String tableName = this._tableName;
128.          String[] columnNames = this._columnNames;
129.
130.          if(mainCursor.first(tableName))
131.          {
132.             IData[] recordIData;
133.             if(this._repeating)
134.             {
135.                recordIData = IDataUtil.getIDataArray (mainCursor,tableName);
136.                data = new Hashtable[recordIData.length];
137.             }
138.             else
139.             {
140.                recordIData = new IData[] {IDataUtil.getIData(mainCursor)};
141.                data = new Hashtable[1];
142.             }
143.             for(int rec=0;rec<recordIData.length;rec++)
144.             {
145.                IDataCursor recordCursor = recordIData[rec].getCursor();
146.                data[rec] = new Hashtable();
147.                for(int c = 0; c < columnNames.length;c++)
148.                {
149.                   if(recordCursor.first(columnNames[c]))
150.                   {
151.                   data[rec].put(tableName + "." + columnNames[c],
152.                      recordCursor.getValue());
153.                   }
154.                }
155.                recordCursor.destroy();
156.                }
157.             }
158.             else
159.             {
160.                throw MyAdapter.getInstance().createAdapterException(9999,
161.                   new String[] {"No Request Data"});
162.             }
163.          }
164.          catch (Throwable t)
165.          {
166.             throw MyAdapter.getInstance().createAdapterException(9999,
167.                new String[] {"Error unpacking request data"},t);
168.          }
169.          finally
170.          {
171.             mainCursor.destroy();
172.          }
173.          return data;
174.       }
175.
176.       private WmRecord packResonse(Hashtable[] response) throws

ResourceException
177.       {
178.          WmRecord data = null;
179.          try
```

```
180.          {
181.             IData[] recordIData = new IData[response.length];
182.             String tableName = this._tableName;
183.             String[] columnNames = this._columnNames;
184.
185.             for(int rec = 0; rec < response.length; rec++)
186.             {
187.                recordIData[rec] = IDataFactory.create();
188.                IDataCursor recordCursor = recordIData[rec].getCursor();
189.                for(int col = 0; col < columnNames.length;col++)
190.                {
191.                   IDataUtil.put(recordCursor,columnNames[col],
192.                         response[rec].get(tableName + "." +
columnNames[col]));
193.                }
194.                recordCursor.destroy();
195.             }
196.             IData mainIData = IDataFactory.create();
197.             IDataCursor mainCursor = mainIData.getCursor();
198.             if(this._repeating)
199.             {
200.                IDataUtil.put(mainCursor,tableName,recordIData);
201.             }
202.             else
203.             {
204.                IDataUtil.put(mainCursor,tableName,recordIData[0]);
205.             }
206.             mainCursor.destroy();
207.             data =
WmRecordFactory.getFactory().createWmRecord("nameNotUsed");
208.             data.setIData(mainIData);
209.          }
210.          catch (Throwable t)
211.          {
212.             throw MyAdapter.getInstance().createAdapterException(9999,
213.             new String[] {"Error packing response data"},t);
214.          }
215.          return data;
216.       }
217.    }
```

Code Example 2: WmManagedConnection Implementation Class Updates

```
1.   private String[] mockTableNames ={ "CUSTOMERS","ORDERS","LINE_ITEMS"};
2.   private String[][] mockColumnNames ={
3.      {"name","id", "ssn"},
4.      {"id","date","customer_id"},
5.      {"order_id","item_number","quantity","description"}
6.     };
7.
8.   private String [][] mockDataTypes = {
9.      {"java.lang.String","java.lang.Integer", "java.lang.String"},
10.     {"java.lang.Integer", "java.util.Date", "java.lang.Integer"},
11.     {"java.lang.Integer", "java.lang.Integer", "java.lang.Integer",
12.                                                "java.lang.String"}
13.    };
14.
```

```
15.
16.  public Boolean adapterCheckValue(String serviceName,
17.                                   String resourceDomainName,
18.                                   String[][] values,
19.                                   String testValue) throws AdapterException
20.  {
21.     Boolean result = new Boolean(false);
22.     if(resourceDomainName.equals(MockDbUpdate.OVERRIDE_TYPES_RD))
23.     {
24.        try
25.        {
26.           Object o = Class.forName(testValue).getConstructor(
27.              new Class[] {String.class}).newInstance(new Object[]{"0"});
28.           result = new Boolean(true);
29.        }
30.        catch (Throwable t){}
31.     }
32.     return result;
33.  }
34.
35.  public ResourceDomainValues[] adapterResourceDomainLookup(String
serviceName,
36.     String resourceDomainName, String[][] values) throws AdapterException
37.  {
38.     ResourceDomainValues[] results = null;
39.     if(resourceDomainName.equals(MockDbUpdate.COLUMN_NAMES_RD)||
40.        resourceDomainName.equals(MockDbUpdate.COLUMN_TYPES_RD))
41.     {
42.        String tableName = values[0][0];
43.        for(int x = 0; x < this.mockTableNames.length;x++)
44.        {
45.           if(this.mockTableNames[x].equals(tableName))
46.           {
47.              ResourceDomainValues columnsRdvs = new ResourceDomainValues(
48.                 MockDbUpdate.COLUMN_NAMES_RD,this.mockColumnNames[x]);
49.              columnsRdvs.setComplete(true);
50.              ResourceDomainValues typesRdvs = new ResourceDomainValues(
51.                 MockDbUpdate.COLUMN_TYPES_RD, this.mockDataTypes[x]);
52.              typesRdvs.setComplete(true);
53.              results = new ResourceDomainValues[] {columnsRdvs,typesRdvs};
54.              break;
55.           }
56.        }
57.     }
58.     else if (resourceDomainName.equals(MockDbUpdate.FIELD_NAMES_RD)||
59.        resourceDomainName.equals(MockDbUpdate.FIELD_TYPES_RD))
60.     {
61.        String tableName = values[0][0];
62.        boolean repeating = Boolean.valueOf(values[1][0]).booleanValue();
63.        String[] columnNames = values[2];
64.        String[] columnTypes = values[3];
65.        String[] overrideTypes = values[4];
66.
67.        String[] fieldNames = new String[columnNames.length];
68.        String[] fieldTypes = new String[columnTypes.length];
69.        String optBrackets;
70.
71.        if(repeating)
72.           optBrackets ="[]";
73.        else
```

```
74.           optBrackets = "";
75.
76.
77.        for (int i = 0; i< fieldNames.length;i++)
78.        {
79.           fieldNames[i] = tableName + optBrackets + "." + columnNames[i];
80.           fieldTypes[i] = columnTypes[i] + optBrackets;
81.
82.           if(overrideTypes.length > i)
83.           {
84.              if (!overrideTypes[i].equals(""))
85.              {
86.                 fieldTypes[i] = overrideTypes[i] + optBrackets;
87.              }
88.
89.           }
90.
91.        }
92.        results = new ResourceDomainValues[]{
93.           new ResourceDomainValues(MockDbUpdate.FIELD_NAMES_RD,fieldNames),
94.           new ResourceDomainValues(MockDbUpdate.FIELD_TYPES_RD,fieldTypes)};
95.
96.     }
97.
98.     return results;
99.  }
100.
101. public void registerResourceDomain(WmAdapterAccess access)
102.    throws AdapterException
103. {
104.    ResourceDomainValues tableRdvs = new ResourceDomainValues(
105.       MockDbUpdate.TABLES_RD,mockTableNames);
106.    tableRdvs.setComplete(true);
107.    access.addResourceDomain(tableRdvs);
108.
109.    access.addResourceDomainLookup(MockDbUpdate.COLUMN_NAMES_RD,this);
110.    access.addResourceDomainLookup(MockDbUpdate.COLUMN_TYPES_RD,this);
111.
112.    ResourceDomainValues rdvs = new ResourceDomainValues(
113.       MockDbUpdate.OVERRIDE_TYPES_RD, new String[] {""});
114.    rdvs.setComplete(false);
115.    rdvs.setCanValidate(true);
116.    access.addResourceDomain(rdvs);
117.    access.addCheckValue(MockDbUpdate.OVERRIDE_TYPES_RD,this);
118.
119.    access.addResourceDomainLookup(MockDbUpdate.FIELD_NAMES_RD,this);
120.    access.addResourceDomainLookup(MockDbUpdate.FIELD_TYPES_RD,this);
121. }
```

Code Example 3: Resource Bundle Updates

```
1.  ,{MockDbUpdate.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
2.     "Mock Update Service"}
3.  ,{MockDbUpdate.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
4.     "Simulates a database update service"}
5.  ,{MockDbUpdate.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL,
6.     "MyAdapter/MockUpdateHelp.txt"}
7.
```

```
8.  ,{MockDbUpdate.UPD_SETTINGS_GRP + ADKGLOBAL.RESOURCEBUNDLEKEY_GROUP,
9.     "Mock Db Update Settings"}
10. ,{MockDbUpdate.UPD_SETTINGS_GRP + ADKGLOBAL.RESOURCEBUNDLEKEY_GROUPURL,
11.    "MyAdapter/UpdateGroupHelp.html"}
12.
13. ,{MockDbUpdate.COLUMN_NAMES_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
14.    "Column Names"}
15. {MockDbUpdate.COLUMN_NAMES_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
16.    "Name of column updated by this service"}
17.
18. ,{MockDbUpdate.COLUMN_TYPES_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
19.   "Column Types"}
20. ,{MockDbUpdate.COLUMN_TYPES_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
21.    "Default data type for column"}
22.
23. ,{MockDbUpdate.OVERRIDE_TYPES_PARM +
ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
24.    "Override Data Types"}
25. ,{MockDbUpdate.OVERRIDE_TYPES_PARM +
ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
26.    "Type to override column default"}
27.
28. ,{MockDbUpdate.REPEATING_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
29.    "Update Multiple Rows?"}
30. ,{MockDbUpdate.REPEATING_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
31.    "Select if input will include multiple rows to update"}
32.
33. ,{MockDbUpdate.TABLE_NAME_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
34.    "Table Name"}
35. ,{MockDbUpdate.TABLE_NAME_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
36.    "Select Table Name"}
```

Configuring and Testing Adapter Service Nodes

Now you are ready to configure an adapter service node as follows:

- Compile your adapter as described in "Compiling the Adapter" on page 43.
- Reload your adapter as described in "Loading, Reloading, and Unloading Packages" on page 152.
- Refresh the Designer cache.
- Configure and enable an adapter service node as described in "Configuring Adapter Service Nodes" on page 156.

6 Polling Notifications

- Overview 108
- Polling Notification Classes 109
- The Metadata Model for Polling Notifications 110
- Polling Notification Callbacks 110
- The runNotification Method 111
- Polling Notification Interactions 112
- Polling Notification Implementation 115
- Cluster Support for Polling Notifications 123

Overview

A polling notification is a facility that enables an adapter to initiate activity on webMethods Integration Server, based on events that occur in the adapter resource. A polling notification monitors an adapter resource for changes (such as an insert, update, or delete operation) so that the appropriate flow or Java services can react to the data, such as sending an invoice or publishing an invoice to Integration Server.

Adapter users create a polling notification node using Software AG Designer. They assign to the notification an adapter connection node that they created earlier. At the same time, Designer creates a document type node that describes the data generated by the polling notification when it executes. The notification publishes this document and sends it to Integration Server. For more information on Integration Server publishable documents, see the *Publish-Subscribe Developer's Guide* for your release.

Important:
If you are using Integration Server 8.0 SP1 or earlier, a polling notification cannot use a connection that is also used for an adapter listener.

To process a document associated with the notification, adapter users can use an Integration Server trigger. When Integration Server receives a document, the trigger invokes the flow or Java service registered with the trigger. The service then processes the data contained in the notification's document.

Finally, adapter users must specify notification scheduling parameters that specify the interval at which Integration Server should invoke the notification, and then enable the notification. To accomplish these tasks, they use Integration Server Administrator. For instructions on creating and using polling notification nodes, see "Polling Notification Nodes" on page 157.

Implementing Polling Notifications

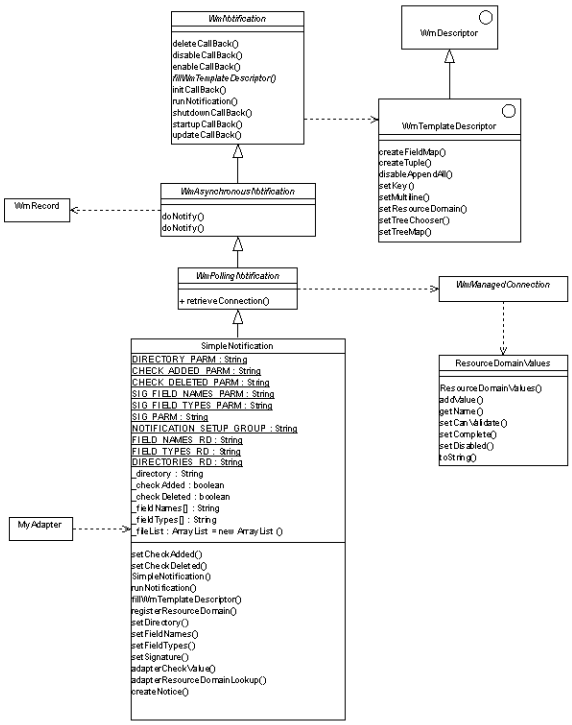
The implementation of a polling notification is similar to the implementation of an adapter service. Each implementation includes a Java class extending an ADK base class, and a namespace node in which design-time configuration data is stored. In the Java class, the metadata model for polling notifications is nearly identical to that of adapter services. The configuration pages are built from the polling notification's metadata.

The primary difference between adapter services and polling notifications is the run time behavior of polling notifications. Polling notifications cannot be directly invoked from a flow service (or from Designer). Instead, the server invokes a polling notification automatically, based on a fixed time interval. When a polling notification determines that a specified event has occurred in the adapter resource, it produces a document describing the event. These documents are automatically published to Integration Server (or webMethods Broker) as they are generated by the notification. The processing of the published document is based on triggers that are configured to invoke flow services when the given document type is published.

Polling Notification Classes

You create an adapter service by extending the base class WmPollingNotification.The following diagram shows the classes provided by the ADK to support polling notifications, and the WmPollingNotification implementation class SampleNotification.

Polling notification classes



In your WmPollingNotification implementation class, implement the following methods:

Method	Description
fillWmTemplateDescriptor	This method serves the same purpose as the WmAdapterService method of the same name. That is, it serves to modify how metadata parameters are handled during data entry. Failing to override fillWmTemplateDescriptor results in a run-time error. For more information, see "The WmTemplateDescriptor Interface" on page 70.
runNotification	Provides the run-time entry point for the notification (see "Defining a WmPollingNotification Implementation Class" on page 116).

Method	Description
Various accessor methods	Optional. Provided to define metadata parameters. Metadata parameters for polling notifications work exactly the same way they do in adapter services.
Callback override methods	Optional. The implementation may override any or all of the callback methods defined in WmNotification. For more information, see "Polling Notification Callbacks" on page 110.
metadataVersion	Returns the current version of the metadata. **Note:** Override this method if the template has multiple metadata versions.
fieldsToIgnoreInMetadataDefinition	Uses metadata version as input and returns an array of the fields that are not applicable to the metadata version provided. **Note:** Override this method if the template has multiple metadata versions. Failing to override fieldsToIgnoreInMetadataDefinition results in breaking old services.

The Metadata Model for Polling Notifications

The metadata model for polling notifications is identical to the model for adapter services, except for the following:

- A polling notification has no input signature. The output signature is constructed in the same way, but the server uses it to generate a document type node that enables triggers to identify notification data in the node.
- You register polling notifications in the WmAdapter.fillAdapterTypeInfo method instead of WmManagedConnectionFactory.fillResourceAdapterMetadataInfo.

Polling Notification Callbacks

The WmNotification base class defines a set of callback methods that you can override in any notification implementation class. The following table describes when these methods are called, and the impact of an exception thrown from the method. For complete details, see the Javadoc for the WmNotification class.

This callback ...	Is called when ...	Exception effect
deleteCallBack	Any attempt is made to delete or rename the notification node.	Error logged, node will not be deleted/renamed.
disableCallBack	The node status is changed to disabled.	Error logged but node is still disabled.
enableCallBack	The node status is changed from disabled to enabled.	Error logged, node status remains disabled.
initCallBack	The node is created, the package is enabled, and so on.	Error logged.
resumeCallBack	The node status is changed from suspended to enabled.	Error logged, node remains suspended.
shutdownCallBack	The node is disabled or suspended, the server is shutdown, the package is disabled, and so on.	Error logged.
startupCallBack	The node is enabled or resumed, the server starts, and so on.	Error logged, node is disabled.
suspendCallBack	The node state is changed from enabled to suspended.	Error logged.
updateCallBack	The node is modified (not called when node is created).	Error logged, updates are discarded.

Note:
In all cases, an AdapterException will cause the associated connection to be destroyed and removed from the pool.

The runNotification Method

The WmNotification.runNotification method is similar to the WmAdapterService.execute method, except that there is no input to this method. Results are delivered by calling the WmAsynchronousNotification.doNotify method, and passing it a WmRecord instance that must conform to the output signature of the polling notification node (see "Specifying Notification Signatures (Document Type)" on page 118). The WmRecord is constructed in exactly the same way it is constructed for adapter services (see "Adapter Service Execution" on page 87).

The Exactly Once Notification Feature

The Exactly Once notification feature ensures that duplicate polling notifications will not be processed, even if a failure occurs during processing.

To use the Exactly Once feature, use the following form of the doNotify method in the notification, and provide a resource-specific msgID value with each notification record:

```
public void doNotify(WmRecord rec, java.lang.String msgID)
```

The adapter implementation must guarantee that the value of msgID is unique and constant for each notification event. (A *notification event* is defined as any activity on the adapter resource that will cause your runNotification implementation to call doNotify.) This means that the msgID will never be duplicated for different notification events, but the msgID will be the same if the same notification event is retrieved multiple times from the adapter resource (even in a failure-recovery scenario).

The server guarantees that msgID values generated by different notification nodes are unique. It accomplishes this by combining the adapter-provided msgID value with a GUID that is created by the server and associated with the notification node when it is created.

Note:
The length (the number of characters) of the value in msgId should not exceed a particular limit. To determine this limit, call the WmAsynchronousNotification.adapterMaxMessageIdLen() method. There is a fixed number of characters available in Integration Server to hold a notification ID. Of these, the WmART package reserves a certain number to hold a unique ID that it inserts prior to dispatching a notification. The remaining characters are available to you when calling WmAsynchronousNotification.doNotify(WmRecord rec, String msgId). For more information, see the Javadoc for WmAsynchronousNotifcation.doNotify.

If you do not want to use the Exactly Once feature, use the following form of doNotify:

```
public void doNotify(WmRecord rec)
```

For more information, see the Javadoc for WmAsynchronousNotification.

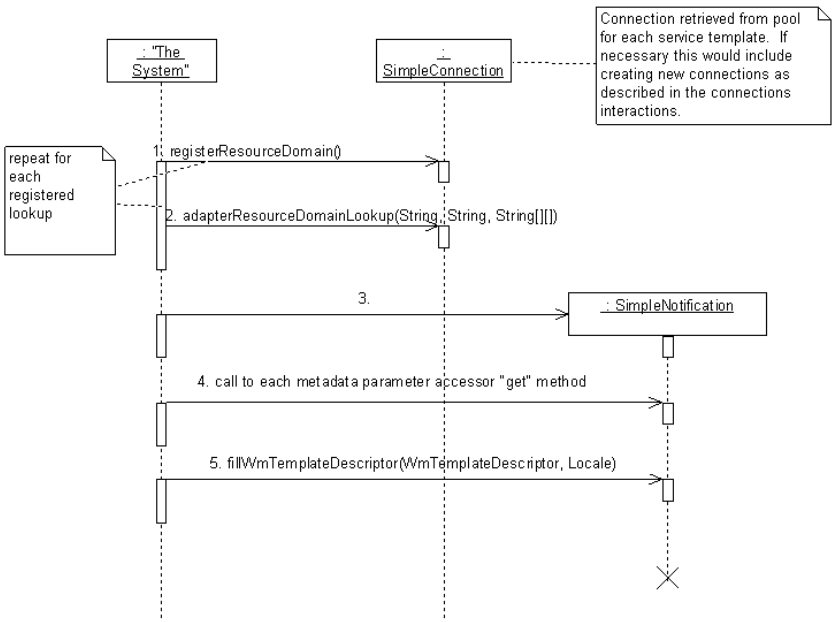
Polling Notification Interactions

Although polling notifications are structurally similar to adapter services, the dynamic model is similar only in the way in which metadata is initialized, as described below.

Loading Polling Notification Templates

As with adapter services, Designer caches metadata values for polling notifications. These values include resource domain values and template descriptor information. The following figure shows the interactions within the adapter as Designer loads its cache for a polling notification. This interaction occurs either when a new polling notification node is created, or an existing one is viewed (if the data is not already held in the Designer cache).

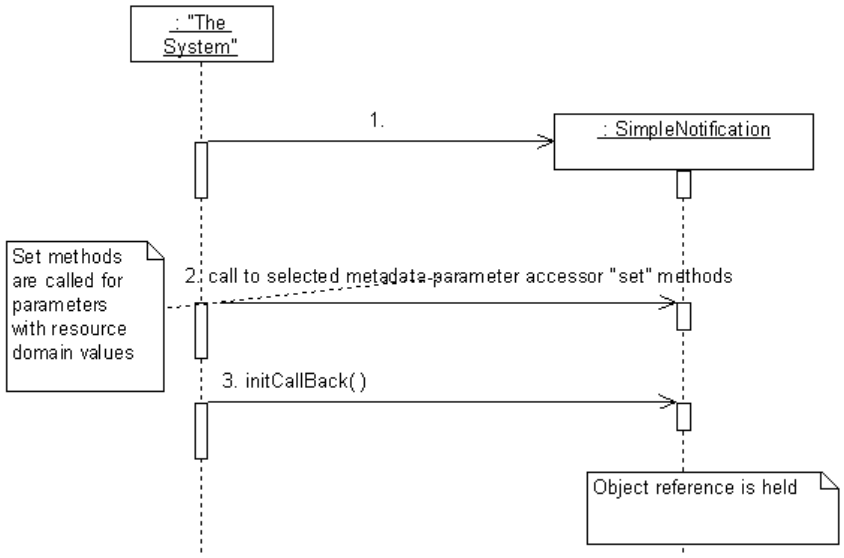
Loading a polling notification template



Creating and Loading Polling Notification Nodes

When an adapter user creates a new polling notification node, or loads an existing node during server startup or package startup, the server instantiates the appropriate class and executes the initCallBack method. The containing package holds the object reference for the lifetime of the containing package. If this interaction is initiated by a package load, and the polling notification node is enabled, the enable/startup interaction occurs immediately afterward.

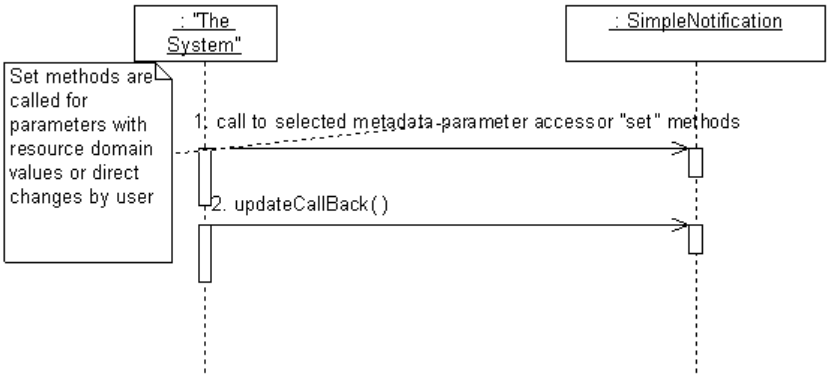
Creating and loading a polling notification



Updating Polling Notifications

Unlike with adapter services, polling notification parameter values are updated each time an adapter user saves the values in Designer. After the "set" methods pass the modified values to the object instance, the notification calls the updateCallBack method. If that method throws an exception, it prevents the values from being persisted in the notification node.

Uploading a polling notification

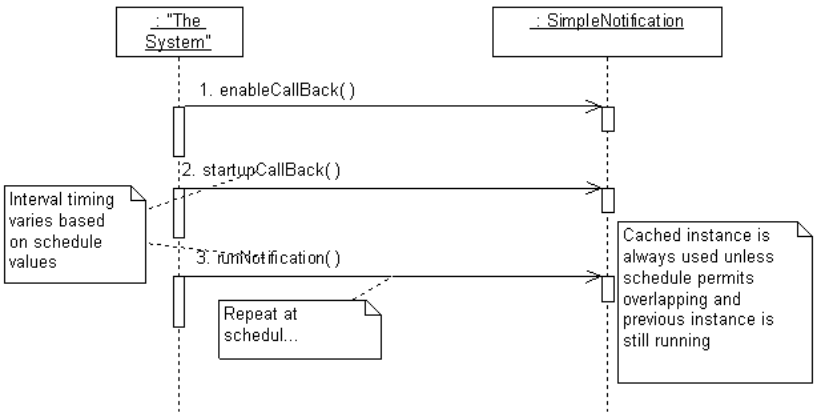


Enabling Polling Notifications

When an adapter user enables a polling notification using Integration Server Administrator, the notification calls the enableCallBack method before the startupCallBack method. If the node was previously enabled, and the user is simply starting up the notification after the package loads, then the enableCallBack call is skipped. An exception from either method call disables the notification node.

The server calls the runNotification method at regular intervals, based on the scheduling parameters that specify the interval at which Integration Server should invoke the notification. The same object instance is always used unless the schedule is configured to allow overlapping, and the previous call to runNotification has not completed.

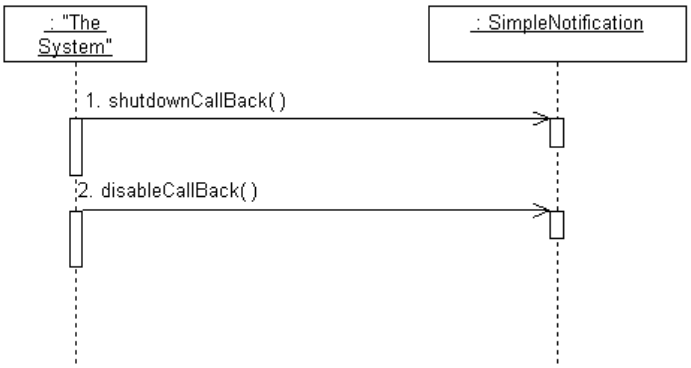
Enabling a polling notification



Disabling Polling Notifications

The following figure shows the interactions that occur when an adapter user explicitly disables a polling notification node. If a node is shut down by any other means, the disableCallBack is skipped.

Disabling a polling notification



Polling Notification Implementation

The tasks for implementing a polling notification are as follows:

- "Defining a WmPollingNotification Implementation Class" on page 116.
- "Updating the fillAdapterTypeInfo Method" on page 116.
- "Updating the Resource Bundle" on page 116.
- "Specifying Configuration Metadata for Polling Notifications" on page 117.
- "Implementing Configuration Resource Domains for Polling Notifications" on page 117.
- "Specifying Notification Signatures (Document Type)" on page 118.
- "Implementing Signature Resource Domains" on page 119.
- "Implementing the runNotification Method and Callbacks" on page 119.
- "Configuring and Testing Polling Notification Nodes" on page 119.

The example polling notification implementation shown in "Example Polling Notification: SimpleNotification" on page 119 monitors the contents of a directory and sends notifications when files are added to, or removed from, the directory. Although this example is very simple, it demonstrates most of the notification capabilities, except for callbacks.

Note:
This example implements a design strategy that enables you to encapsulate the resource domain support inside an adapter service or notification. This strategy is discussed in "An Alternative Approach to Organizing Resource Domains" on page 176. You do not need to fully understand this strategy to understand the example code. However, if you are uncomfortable with this

strategy, you may implement those methods in your connection implementation. You will have to adjust the method signatures and the "this" references appropriately. The "this" reference refers to the notification. If you move the methods to the connection, then the "this" refers to the connection.

Defining a WmPollingNotification Implementation Class

Create a class that extends com.wm.adk.notification.WmPollingNotification, as shown in "Example Polling Notification: SimpleNotification" on page 119, line 18. The class must implement the abstract methods runNotification and fillWmTemplateDescriptor (Lines 56 and 103). Both methods may be empty initially. For more information, see "Implementing the runNotification Method and Callbacks" on page 119.

Updating the fillAdapterTypeInfo Method

Update the fillAdapterTypeInfo method of your adapter definition class (the WmAdapter implementation class) to register your polling notification type.

For example:

```
public void fillAdapterTypeInfo(AdapterTypeInfo info, Locale locale)
{
   info.addNotificationType(SimpleNotification.class.getName());
}
```

Updating the Resource Bundle

Optionally, update the resource bundle with display names, descriptions, and a help URL to make the notification more usable, as follows:

```
,{SimpleNotification.class.getName() +
ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
    "Simple Polling Notification"}
,{SimpleNotification.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
    "Looks for file updates to a specified directory"}
,{SimpleNotification.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL,
   "MyAdapter/SimpleNotificationHelp.txt"}
,{SimpleNotification.CHECK_ADDED_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
   "Notify on Add"}
,{SimpleNotification.CHECK_ADDED_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
   "Check if notification should be generated when file added"}
,{SimpleNotification.CHECK_DELETED_PARM +
ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
   "Notify on Delete"}
,{SimpleNotification.CHECK_DELETED_PARM +
ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
   "Check if notification should be generated when file deleted"}
,{SimpleNotification.DIRECTORY_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
   "Directory Path"}
,{SimpleNotification.DIRECTORY_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
   "Directory to monitor"}
,{SimpleNotification.NOTIFICATION_SETUP_GROUP +
ADKGLOBAL.RESOURCEBUNDLEKEY_GROUP,
```

```
   "Simple Notification Settings"}
,{SimpleNotification.NOTIFICATION_SETUP_GROUP +
```

```
ADKGLOBAL.RESOURCEBUNDLEKEY_GROUPURL,
   "MyAdapter/SimpleNotificationGroupHelp.html"}
```

Specifying Configuration Metadata for Polling Notifications

The next step for implementing a polling notification is to create the metadata constructs that users will use for entering data when they create polling notification nodes. To do this, you create metadata parameters appropriate to the function of the polling notification, and describe display and data entry attributes for those parameters.

The example implementation includes three metadata parameters:

Parameter	Description
directory	The directory to monitor.
checkAdded	Indicates whether notifications should be sent when files are added to the specified directory.
checkDeleted	Indicates whether notifications should be sent when files are deleted from the specified directory.

Each parameter has:

- An accessor method (see "Example Polling Notification: SimpleNotification" on page 119, lines 47-49).
- A variable to hold the configured values (lines 38-40).
- A String constant containing the name of the parameter (lines 22-24).
- A set of resource bundle entries with a localizable parameter name and description (as shown previously, in "Updating the Resource Bundle" on page 116).

For more information on metadata parameters, see "webMethods Metadata Parameters" on page 58.

Next, specify the parameters' display and data entry attributes by calling various methods of the WmTemplateDescriptor interface from the service's fillWmTemplateDescriptor method. In the example code, each data entry parameter is placed (in display order) in a single group referenced by the constant NOTIFICATION_SETUP_GROUP. (A constant (instead of a string) names the group because the same value is used in the resource bundle to specify a localizable group name and help URL.)

Implementing Configuration Resource Domains for Polling Notifications

The next steps for implementing a polling notification are:

- Defining and implementing the resource domains required for the metadata parameters you created in "Specifying Configuration Metadata for Polling Notifications" on page 117.
- Identifying the values upon which those resource domains depend.

For each parameter that requires a resource domain to supply a value, or that requires a validity check for values supplied by adapter users, you must:

- Call WmTemplateDescriptor.setResourceDomain, passing the name of the parameter, the name of the resource domain, and an array of the names of any parameters on which the resource domain will depend (see "Example Polling Notification: SimpleNotification" on page 119, line 123).
- Register the resource domain support in the registerResourceDomain method of the associated connection class (see lines 195-200).
- Implement code to populate the resource domain values and/or adapter check values (discussed below).

In this example, there are no preset values, but adapter user-supplied data for the directory parameter is validated to ensure that the directory exists (see lines 133-157).

Specifying Notification Signatures (Document Type)

After you implement the configuration logic for the polling notification, you implement logic that defines the signature of the polling notification node. Remember that a polling notification only has an output signature. It is used to create a document type node that enables triggers to identify notification data in the node. To define an output signature, you create additional metadata parameters as follows:

Parameter	Description
signature	Used with the reserved signature resource domain.
fieldName and fieldType	The dependency parameters in which you build the signature data. The relationship between these parameters is established in the WmTemplateDescriptor (as shown in "Example Polling Notification: SimpleNotification" on page 119, lines 117-130). For more information about the mechanics of signature construction, see "Adapter Service Node Signatures" on page 84.

Manipulating Adapter Notification Document Properties

With few exceptions, the document properties for an adapter's polling and listener notifications are managed and manipulated the same way as they are for an adapter service's signature. For the three template-based features (signature wrapping, override connection name, and pass full pipeline), only the pass full pipeline feature applies to notification documents, and then, only for synchronous notifications. When the pass full pipeline option is enabled for a synchronous listener notification, then the notification will be able to pass fields to the invoked service that are not defined in its request document, and receive fields from that service that are not defined in the

notification's reply document. Template-based signature manipulation features have no other effect on notification documents.

Document field properties are managed in exactly the same way as the signature field properties described above. In the case of notifications, the setSignatureProperties() method is called setDocumentProperties(). For asynchronous notifications, there is only one PipelineRecordProperties argument.

Implementing Signature Resource Domains

The resource domain implementation for the signature parameters of this example is straightforward. The signature in this case is static, but it is implemented as a lookup, to facilitate maintenance of the class (see "Example Polling Notification: SimpleNotification" on page 119, lines 167-179).

Implementing the runNotification Method and Callbacks

The final task for implementing a polling notification is to add the runNotification method and any callback methods. This example implements some very basic logic as previously described. It relies on the fact that the object instance is reused between runNotification calls. This may not be a good technique if the runNotification call runs for a long time or if overlapping calls occur. A more robust model would probably use a persistent store instead of an instance variable to track the current directory snapshot.

Configuring and Testing Polling Notification Nodes

Now you are ready to configure a polling notification node as follows:

- Compile your adapter as described in "Compiling the Adapter" on page 43.
- Reload your adapter as described in "Loading, Reloading, and Unloading Packages" on page 152.
- Refresh the Designer cache.
- Configure and enable a polling notification node as described in "Polling Notification Nodes" on page 157.

Example Polling Notification: SimpleNotification

```
1.   package com.mycompany.adapter.myadapter.notifications;
2.
3.   import java.io.File;
4.   import java.util.ArrayList;
5.   import java.util.Locale;
6.   import javax.resource.ResourceException;
7.
8.   import com.mycompany.adapter.myadapter.services.ResourceDomainHandler;
9.   import com.wm.adk.cci.record.WmRecord;
10.  import com.wm.adk.cci.record.WmRecordFactory;
```

```
11.  import com.wm.adk.connection.WmManagedConnection;
12.  import com.wm.adk.error.AdapterException;
13.  import com.wm.adk.metadata.ResourceDomainValues;
14.  import com.wm.adk.metadata.WmAdapterAccess;
15.  import com.wm.adk.metadata.WmTemplateDescriptor;
16.  import com.wm.adk.notification.WmPollingNotification;
17.
18.  public class SimpleNotification extends WmPollingNotification
19.     implements ResourceDomainHandler
20.  {
21.
22.     public static final String DIRECTORY_PARM = "directory";
23.     public static final String CHECK_ADDED_PARM = "checkAdded";
24.     public static final String CHECK_DELETED_PARM = "checkDeleted";
25.
26.     public static final String SIG_FIELD_NAMES_PARM = "fieldNames";
27.     public static final String SIG_FIELD_TYPES_PARM = "fieldTypes";
28.     public static final String SIG_PARM = "signature";
29.
30.     public static final String NOTIFICATION_SETUP_GROUP =
31.        "SimpleNotification.setup";
32.
33.     public static final String FIELD_NAMES_RD =
               "SimpleNotification.fieldNames.rd";
34.     public static final String FIELD_TYPES_RD =
               "SimpleNotification.fieldTypes.rd";
35.     public static final String DIRECTORIES_RD =
36.            "SimpleNotification.direcotries.rd";
37.
38.     private String _directory;
39.     private boolean _checkAdded;
40.     private boolean _checkDeleted;
41.     private String[] _fieldNames;
42.     private String[] _fieldTypes;
43.
44.
45.     private ArrayList _fileList = new ArrayList();
46.
47.     public void setDirectory(String val){_directory = val;}
48.     public void setCheckAdded(boolean val){_checkAdded = val;}
49.     public void setCheckDeleted(boolean val){_checkDeleted = val;}
50.     public void setFieldNames(String[] val){_fieldNames = val;}
51.     public void setFieldTypes(String[] val){_fieldTypes = val;}
52.     public void setSignature(String[] val){}
53.
54.     public SimpleNotification(){}
55.
56.     public void runNotification() throws ResourceException
57.     {
58.        File thisDir = new File(_directory);
59.        File [] newList = thisDir.listFiles();
60.        ArrayList scratchCopy = new ArrayList(this._fileList);
61.
62.        for (int nlIndex = 0;nlIndex < newList.length;nlIndex++)
63.        {
64.           String name = newList[nlIndex].getName();
65.           if(newList[nlIndex].isFile())
66.           {
67.              if(scratchCopy.contains(name))
68.              {
```

```
69.                scratchCopy.remove(name);
70.             }
71.             else
72.             {
73.                this._fileList.add(name);
74.                if(this._checkAdded)
75.                {
76.                   this.doNotify(createNotice(name,_directory,true,false));
77.                }
78.             }
79.          }
80.          else
81.          {
82.             scratchCopy.remove(name);
83.          }
84.
85.       }
86.       // now anything left in the scratch copy is missing from the directory
87.
88.       String[] deadList = new String[scratchCopy.size()];
89.       scratchCopy.toArray(deadList);
90.       for(int dlIndex = 0; dlIndex < deadList.length;dlIndex++)
91.       {
92.          this._fileList.remove(deadList[dlIndex]);
93.          if(this._checkDeleted)
94.          {
95.             this.doNotify(createNotice(deadList[dlIndex],
                                 _directory,false,true));
96.          }
97.
98.       }
99.
100.   }
101.
102.
103.   public void fillWmTemplateDescriptor(WmTemplateDescriptor descriptor
                                                                  Locale l)
104.      throws ResourceException
105.   {
106.      descriptor.createGroup(NOTIFICATION_SETUP_GROUP,
107.         new String[]{DIRECTORY_PARM,
108.                      CHECK_ADDED_PARM,
109.                      CHECK_DELETED_PARM,
110.                      SIG_FIELD_NAMES_PARM,
111.                      SIG_FIELD_TYPES_PARM,
112.                      SIG_PARM});
113.      descriptor.createFieldMap(new String[]
114.         {SIG_FIELD_NAMES_PARM,
115.          SIG_FIELD_TYPES_PARM,
116.          SIG_PARM},false);
117.      descriptor.setHidden(SIG_FIELD_NAMES_PARM);
118.      descriptor.setHidden(SIG_FIELD_TYPES_PARM);
119.      descriptor.setHidden(SIG_PARM);
120.      descriptor.createTuple(
121.         new String[]{SIG_FIELD_NAMES_PARM,SIG_FIELD_TYPES_PARM});
122.
123.      descriptor.setResourceDomain(DIRECTORY_PARM,DIRECTORIES_RD,null);
124.
125.      descriptor.setResourceDomain(SIG_FIELD_NAMES_PARM,FIELD_NAMES_RD,null);
126.      descriptor.setResourceDomain(SIG_FIELD_TYPES_PARM,FIELD_TYPES_RD,null);
```

```
127.       descriptor.setResourceDomain(SIG_PARM,
128.          WmTemplateDescriptor.OUTPUT_FIELD_NAMES,
129.          new String[]{SIG_FIELD_NAMES_PARM,SIG_FIELD_TYPES_PARM});
130.       descriptor.setDescriptions(
              MyAdapter.getInstance().getAdapterResourceBundleManager(),l);
131.    }
132.
133.    public Boolean adapterCheckValue(
134.       WmManagedConnection connection,
135.       String resourceDomainName,
136.       String[][] values,
137.       String testValue)
138.       throws AdapterException
139.    {
140.
141.       boolean result = true;
142.
143.       if(resourceDomainName.equals(DIRECTORIES_RD))
144.       {
145.          File testDir = new File(testValue);
146.          if (!testDir.exists())
147.          {
148.             result = false;
149.          }
150.          else if(!testDir.isDirectory())
151.          {
152.             result = false;
153.          }
154.       }
155.
156.       return new Boolean(result);
157.    }
158.
159.    public ResourceDomainValues[] adapterResourceDomainLookup(
160.       WmManagedConnection connection,
161.       String resourceDomainName,
162.       String[][] values)
163.       throws AdapterException
164.    {
165.       ResourceDomainValues[] results = null;
166.
167.       if (resourceDomainName.equals(FIELD_NAMES_RD)
168.          || resourceDomainName.equals(FIELD_TYPES_RD))
169.       {
170.          ResourceDomainValues names =
171.             new ResourceDomainValues(FIELD_NAMES_RD,new String[] {
172.             "FileName", "Path","isAdded","isDeleted"});
173.
174.          ResourceDomainValues types =
175.             new ResourceDomainValues(FIELD_TYPES_RD,new String[] {
176.             "java.lang.String", "java.lang.String",
177.             "java.lang.Boolean","java.lang.Boolean"});
178.          results = new ResourceDomainValues[] {names,types};
179.       }
180.
181.       return results;
182.    }
183.
184.    public void registerResourceDomain(
185.       WmManagedConnection connection,
```

```
186.       WmAdapterAccess access)
187.       throws AdapterException
188.    {
189.
190.       access.addResourceDomainLookup(this.getClass().getName(),
191.          FIELD_NAMES_RD,connection);
192.       access.addResourceDomainLookup(this.getClass().getName(),
193.          FIELD_TYPES_RD,connection);
194.
195.       ResourceDomainValues rd = new ResourceDomainValues(DIRECTORIES_RD,
196.          new String[] {""});
197.       rd.setComplete(false);
198.       rd.setCanValidate(true);
199.       access.addResourceDomain(rd);
200.       access.addCheckValue( DIRECTORIES_RD,connection);
201.
202.    }
203.
204.    private WmRecord createNotice(String file,
205.       String dir, boolean isAdded, boolean isDeleted)
206.    {
207.      WmRecord notice =
WmRecordFactory.getFactory().createWmRecord("notUsed");
208.      notice.put("FileName",file);
209.      notice.put("Path",dir);
210.      notice.put("isAdded",new Boolean(isAdded));
211.      notice.put("isDeleted", new Boolean(isDeleted));
212.
213.       return notice;
214.
215.    }
216.
217. }
```

Cluster Support for Polling Notifications

When Integration Servers are deployed in a cluster, the servers in that cluster automatically share information about the registered polling notifications. The adapter run time will automatically coordinate polling and some callbacks among instances of the same polling notification node on different servers in a cluster.

This coordination of clustered polling notifications requires no special coding by the adapter. However, there are design considerations for adapters that will be used in a cluster. There are also global, adapter-specific, and node-specific configuration options that control how coordination is performed. At a minimum, adapter writers should specify configuration values that are appropriate for the adapter.

Callback Coordination

When a callback is coordinated across the cluster, that callback is only executed on one instance of the polling notification in the cluster. For example, if the enableCallBack is coordinated, the first polling notification in the cluster to be enabled will execute the enableCallBack. When subsequent instances of that notification are enabled, the enableCallBack call is suppressed. If the

callbacks were not coordinated, each instance would execute the enableCallBack when the node instance was enabled.

The purpose of callback coordination is to prevent redundant updates to the backend associated with starting or stopping a notification. For example, when the disableCallBack is called on a polling notification for the webMethods JDBC Adapter, a database trigger that gathers information for the notification is removed. Without coordination, all instances of that notification would be effectively disabled as soon as the first instance was disabled. With coordination, the disableCallBack is not executed until the last instance of that notification in the cluster is disabled.

From a design standpoint, it is important to segregate management of resources on the backend in separate callbacks from management of resources that are local to the notification instance. Callback coordination is configured so that related pairs of callbacks are either coordinated, or not. The following pairs of callbacks may be coordinated:

- **Enable/Disable**. This occurs when the persistent node state is changed from or to "disabled". When coordinated, the first instance to go from "disabled" to "enabled" will execute the enableCallBack and the last instance to go from either "suspended" or "enabled" to "disabled" executes the disableCallBack.
- **Startup/Shutdown**. This occurs when the node becomes active or inactive for any reason, including when the node is enabled, disabled, suspended, or resumed (if the node is enabled, it also includes server and package startup or shutdown). When coordinated, the first instance to become active will execute the startupCallBack and the last instance to become inactive executes the shutdownCallBack.
- **Suspend/Resume**. This occurs when the node is suspended or resumed. When coordinated, the first instance to go from "enabled" to "suspended" will execute the suspendCallBack and the last instance to go from either "suspended" to "enabled" executes the resumeCallBack.

For information on how to configure callback coordination, see "Configuration Settings" on page 126.

Polling Coordination

When a polling notification is started, it executes polls on an interval according to its schedule configuration. When polling is coordinated across a cluster, a poll is executed on only one of the active instances at each scheduled interval. Which instance executes depends on configuration and timing.

Integration Server supports a coordination mode configuration setting for each polling notification node. This configuration is normally set from Integration Server Administrator on the same page where the schedule is set. The following coordination mode values are supported:

Coordination Mode Setting	Description
Disabled	Disables all cluster coordination for this node (both polling and callback coordination). Instances of this node will act entirely independently with no consideration for the cluster.

Coordination Mode Setting	Description
Standby	The first instance of this notification to start will execute all polls until that instance either shuts down, or fails. When that occurs, the first active notification instance that detects that the original instance is no longer polling will take its place.
Distributed	Behaves much like standby, except that at the end of each interval, the instance that first detects that the time to poll has arrived will execute the poll. When the server clocks are properly synchronized, generally the server with the lightest load will execute the poll.

Starting with Integration Server 8.0, the coordination information for clustered polling notifications is stored in the cluster's shared cache.

All coordination of clustered polling notifications is done through an entry specific to the node in the cluster's shared cache. When the first instance of a notification (with coordination enabled) is introduced into a cluster, a new shared cache session is created and populated with the name of the server hosting the node, and the state of the node. As that node is copied to other servers in the cluster, each instance is registered in the same shared cache session. The states of these respective instances are used to determine when coordinated callbacks should be executed.

Important:
Always copy polling notification nodes instead of creating new nodes with the same name and configuration. Each polling notification node is created with a GUID that forms part of the message ID of all documents published by the notification. If the instances do not have the same GUID, it can interfere with duplicate-message detection facilities.

When the first instance of a clustered polling notification is started, that instance is marked as "primary", its schedule and coordination settings are recorded, and the time it calculates for the next poll are all recorded in the shared cache session. Being "primary" means that first instance will execute the first poll (barring failures). When it schedules the next poll, it will release the "primary" status if coordination is distributed, or retain it if coordination is configured for standby mode.

When another instance of the notification starts, it first detects that a previous instance was already started. Since it is not first, it overwrites its own cluster and schedule settings with those recorded in the shared cache. It then schedules itself to "wake up" at the next scheduled poll time. When it wakes up, if another instance is marked as primary, the notification instance will verify that the indicated instance is still active, then reschedule itself to wake up periodically until it detects that the poll was completed within the configured time limit. It then reschedules itself to wake up at the next scheduled poll and repeats the process. If the polling time arrives and no instance is marked as primary, or it detects that the primary instance is no longer functioning, then the local instance assumes the primary role and executes the poll as described above. Since all coordination is based on timestamps recorded in the shared cache, it is very important for server clocks to be synchronized.

Configuration Settings

Cluster coordination is controlled by a number of configuration settings to control behavior and tune failure detection using timeouts.

Global Settings

The following settings are set in the server.cnf file and apply globally to all clustered notifications for all adapters:

- **watt.art.clusteredPollingNotification.keepAliveInterval** - frequency, in milliseconds, with which a secondary instance will check to see if an executing instance is still alive. If not set, the secondary instance will change to the default maxLockDuration value of "180000" for the shared cache.
- **watt.art.clusteredPollingNotification.keepAliveExpireTimeout** - amount of time, in milliseconds, that an executing node can be late before it is assumed to have failed. In general, this setting should be equal to the amount of drift anticipated on the server clocks. If not set, the secondary instance will change to the default maxLockDuration value of "180000" for the shared cache.

Adapter-Specific Settings

Within the configuration directory of the adapter's package, the clusterProperties.cnf provides settings that specify a callback scheme, and place limits on which coordination modes can be applied to notification nodes for the adapter. The clusterProperties.cnf file is an XML file in which settings may be provided globally for the adapter or specifically to a particular notification template. Template-specific settings use the template class name to set the scope of the setting.

The following example includes all of the major constructs of a clusterProperties.cnf file:

```
<?xml version="1.0"?>
<clusterProps>
  <pollingNotifications>
    <callbackScheme>1</callbackScheme>
    <runtimeModeLimit>distribute</runtimeModeLimit>
    <template className="com.wm.adapter.wmarttest.notification.LatchedPollingNot
    ification">
       <callbackScheme>1</callbackScheme>
       <runtimeModeLimit>standby</runtimeModeLimit>
    </template>
  </pollingNotifications>
  <listenerNotifications>
    <callbackScheme>1</callbackScheme>
  </listenerNotifications>
  <listeners>
    <runtimeModeLimit>standby</runtimeModeLimit>
  </listeners>
</clusterProps>
```

The outer <clusterProps> wrapper contains the following three elements:

- <pollingNotifications> wraps settings for clustered polling notifications.

- <listenerNotifications> is for future use. Copy wrapper and contents from the example.
- <listeners> is for future use. Copy wrapper and contents from the example.

The <pollingNotifications> wrapper should contain a global <callbackScheme> and a <runtimeModeLimit> setting that provide the adapter's default values. These can be followed by any number of template wrappers, which contain the <callbackScheme> and <runtimeModeLimit> settings specific to each template. Template-specific settings are applied to notification nodes created from the associated template name.

The <callbackScheme> setting controls how callback coordination is performed, while <runtimeModeLimit> constrains the coordination mode setting that can be set for a notification node. Valid values for these settings are included in the following tables.

Values for callbackScheme Setting

When callbackScheme is set to...	The following Coordination Modes are set:		
	Enable/Disable	**Startup/Shutdown**	**Resume/Suspend**
0	No Coordination	No Coordination	No Coordination
1(default)	Coordinated	No Coordination	No Coordination
2	No Coordination	Coordinated	No Coordination
3	Coordinated	Coordinated	No Coordination
4	No Coordination	No Coordination	Coordinated
5	No Coordination	Coordinated	Coordinated
6	Coordinated	No Coordination	Coordinated
7	Coordinated	Coordinated	Coordinated

Values for runtimeModeLimit Setting

runtimeModelLimit Value	Result
disable	Nodes created using this template cannot be coordinated across a cluster. Nodes will be forced to coordination mode "disabled".
standby (default)	Nodes can be coordinated in "standby" mode or coordination may be disabled.
distribute	Nodes can be coordinated in "distributed" or "standby" mode, or coordination may be disabled.

When a polling notification is created or registered on a cluster-aware Integration Server, the Integration Server will look for a clusterProperties.cnf file in the adapter's config directory. If the file contains no entry for the notification's template, a new <template> entry is created using the

settings specified globally for all polling notifications. If the file is completely absent or unreadable, the file will be created using the default settings identified above.

Node-Specific Settings

The polling notification schedule page in Integration Server Administrator includes cluster-settings that are only editable in a clustered environment. On this page, the coordination mode may be set to one of the values supported by its template (see runtimeModeLimit above). In addition, timeouts may be separately configured for polling and setup (callback) operations.

7 Listener Notifications

- Overview 130
- Listener Classes 131
- Asynchronous Listener Notification Classes 133
- Synchronous Listener Notification Classes 134
- Listener and Listener Notification Interactions 135
- Listener Implementation 138
- Listener Notification Implementation 142

Overview

Listeners and listener notifications work together to create a much more powerful model for detecting and processing events in the adapter resource than is possible with polling notifications.

When the adapter user enables the polling notification node, the server instantiates and initializes a polling notification object with settings from the node. The server then invokes the object (via a call to its runNotification method) on a periodic basis. The polling notification object must retrieve a connection and use it to determine if publishable events have occurred in the adapter resource. If an event has occurred, the polling notification object must generate a WmRecord object conforming to the output signature of the polling notification node, and publish it by calling the doNotify method. Then the polling notification object "goes to sleep" until the next time it is invoked. If the overlap option is enabled in the notification schedule, a second object may be instantiated while a previous instance is still processing events detected in a previous invocation. The second instance retrieves another connection, and interrogates the resource again. This model can make state management significantly more difficult.

With a listener notification, the responsibility for monitoring the adapter resource and processing any events is divided between a listener and its notification(s). A listener object is instantiated and is given a connection when the adapter user enables the associated node. The listener object remains active with the same connection to monitor the resource activity until it is disabled (either explicitly or by disabling the containing package, the adapter, the connection, or webMethods Integration Server). When the listener detects a publishable event in the resource, it passes an object back to the server. The server will interrogate a configured list of listener notifications associated with the listener node until it finds a listener notification node that can process the event. To do this, it calls the listener notification's supports method. The first notification to return true from this call will be invoked using its runNotification method. This behavior is similar to a polling notification in that any data about the event that was provided by the listener is passed as an argument to runNotification.

Important:
If you are using Integration Server 8.0 SP1 or earlier, you must use separate connections for listeners and adapter services as well as listeners and polling notifications.

Synchronous and Asynchronous Listener Notifications

The ADK includes both a synchronous and an asynchronous processing model. An asynchronous listener notification publishes a document to a webMethods Broker queue, using the doNotify method. Adapter users may process the document's data any way they want to. For example, they can create an Integration Server trigger that receives the document and executes an Integration Server flow service or a Java service.

A synchronous listener notification invokes a specified IS service, and potentially receives a reply from the service and delivers the results back to the adapter resource. In this case, the notification object's runNotification method calls invokeService (instead of doNotify), to process the data produced by the notification. A synchronous listener notification does not publish a document.

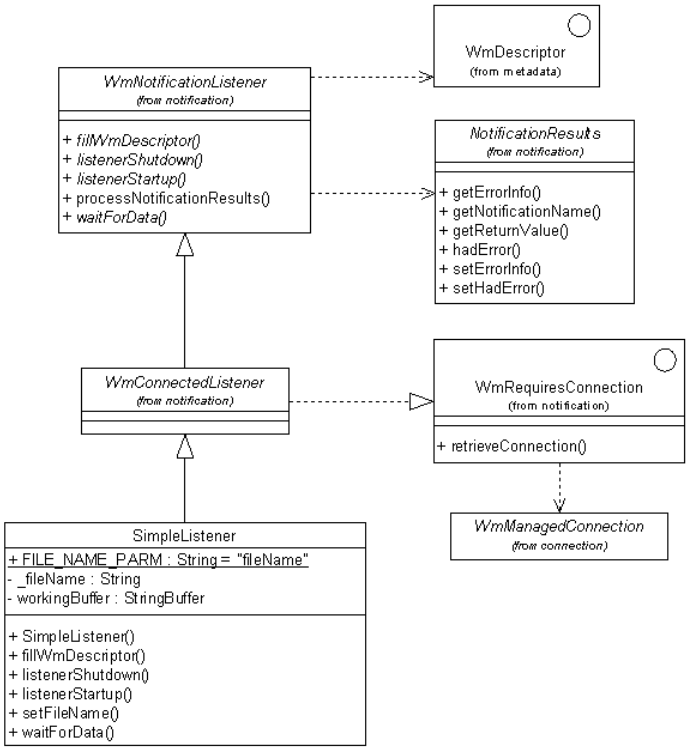
Synchronous listener notifications do not support session handling. When a synchronous listener notification calls a service that needs information contained in the session data, that service can

fail. However, note that the same service may appear to work for an asynchronous listener notification. This is because asynchronous listener notifications themselves do not execute a service. Instead, an Integration Server trigger, which supports session handling, is used to receive the document and execute an Integration Server flow service or a Java service.

Listener Classes

You implement a listener class by extending the base class com.wm.adk.notification.WmConnectedListener, as shown by the SimpleListener class in the following figure.

Listener classes



Like other adapter components, the adapter's implementation class must support both design-time and run-time activities. Metadata parameters for listeners work exactly the same way as described for connections in "webMethods Metadata Parameters" on page 58. In addition, the implementation class is required to override the following methods:

Method	Description
fillWmDescriptor	Adds any display and data entry constraints to the listener's metadata parameters. From the standpoint of the adapter implementation, the model is identical to the connection model.
listenerStartup	Initializes the listener. This method is called during the listener startup sequence as well as during the

Method	Description
	recovery procedure after an AdapterConnectionException is encountered.
waitForData	Monitors the adapter resource. This method returns data that is analyzed by the supports method of associated listener notifications. For more details, see "Listener and Listener Notification Interactions" on page 135.
listenerShutdown	Cleans up listener resources. This method is called during the listener shutdown sequence. **Note:** To allow the listener implementation class to post-process listener notification results, you may optionally override the processNotificationResults method. For an example of using both methods, see "Implementing Run-Time Code" on page 141.
metadataVersion	Returns the current version of the metadata. **Note:** Override this method if the template has multiple metadata versions.
fieldsToIgnoreInMetadataDefinition	Uses metadata version as input and returns an array of the fields that are not applicable to the metadata version provided. **Note:** Override this method if the template has multiple metadata versions. Failing to override fieldsToIgnoreInMetadataDefinition results in breaking old services.

In addition, the implementation class may override the following optional methods:

Method	Description
restrictNotificationTypes	Allows the listener implementation class to restrict the notification classes it supports by explicitly identifying them. For more information, see "Restricting Listeners to Register Specified Notification Templates" on page 140.
shutdownCallBack	Invoked on a thread separate from the listener's thread, this method allows the listener's waitForData loop to be gracefully interrupted prior to a normal shutdown. For more information, see "Implementing the shutdownCallBack() Method" on page 139.

Asynchronous Listener Notification Classes

You implement an asynchronous listener notification class by extending com.wm.adk.notification.WmAsyncListenerNotification, as shown by the SessionLogListenerNotification class in the following figure. As previously mentioned, the implementation of an asynchronous listener notification is similar to a polling notification. The key differences occur in the following methods:

- boolean supports(Object data)

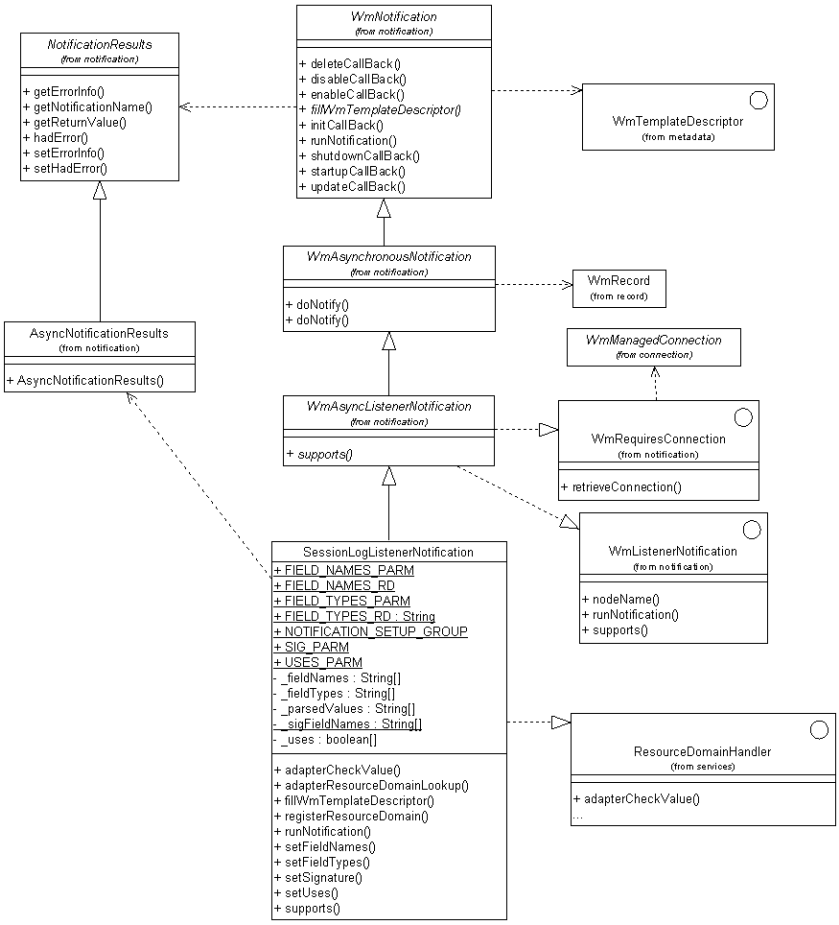
  Determines whether the notification can process the data returned by the listener's waitForData method.

- NotificationResults runNotification(NotificationEvent data)

  If the supports method returns true, the server calls this method (instead of the void runNotification method used in polling notifications) to process the data. The implementation of this method should call one of the WmAsynchronousNotification.doNotify methods for each notification it wants to generate, based on the NotificationEvent data content. This method should never return null. (NotificationEvent simply wraps the data object returned from the listener's waitForData method.)

The SessionLogListenerNotification also implements the ResourceDomainHandler interface, which demonstrates an alternative method of managing resource domains. The ResourceDomainHandler interface is described in "An Alternative Approach to Organizing Resource Domains" on page 176.

Asynchronous listener notification classes



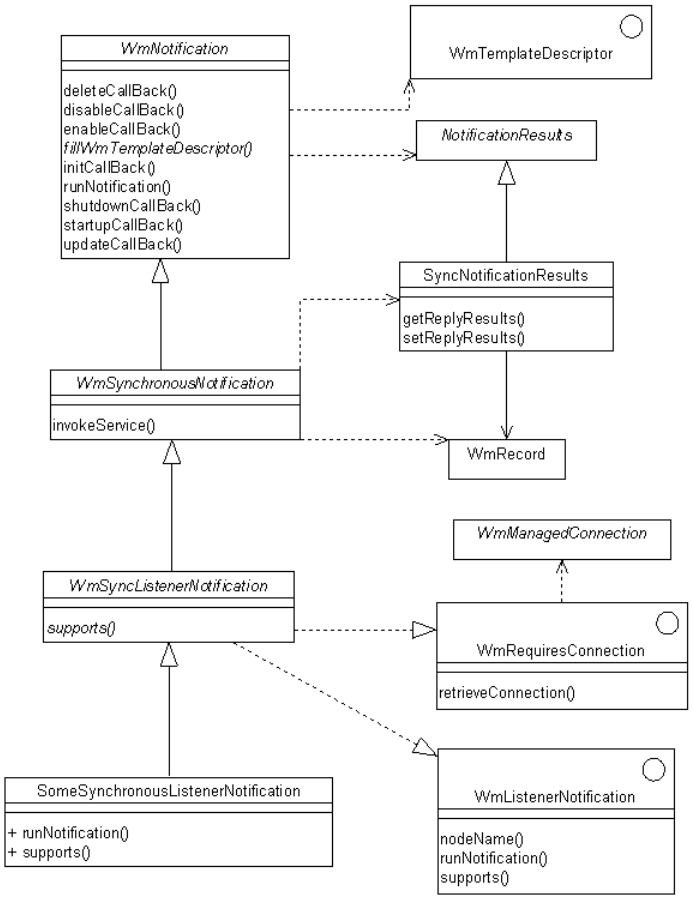
Synchronous Listener Notification Classes

You implement a synchronous listener notification class by extending com.wm.adk.notification.WmSyncListenerNotification. (Examples in this section do not implement a synchronous notification, so the figure in this section only shows a placeholder class.) Unlike the other types of notifications discussed in this document, a synchronous notification does not call doNotify to publish a document. Instead, it calls the WmSynchronousNotification.invokeService method, passing it a WmRecord containing data consistent with the output signature of the notification node.

Synchronous notifications are expected to define both an input and output signature. The terms input and output are relative to the notification. The output signature specifies the format of the data that the notification places on the pipeline prior to invoking the service associated with the synchronous notification. The input signature describes the data that the notification expects to find on the pipeline after the invoked service has completed processing.

The service to be invoked by WmSynchronousNotification.invokeService is specified at design time in the notification node data. When this service is invoked, it executes on a separate thread (and therefore in a different transactional context) from the listener. When the invoked service completes, the server will extract any data on the pipeline that is identified in the notification node's input signature and will deliver that data as a WmRecord wrapped in the SyncNotificationResults object returned by invokeService.

Synchronous listener notification classes



Listener and Listener Notification Interactions

The design-time interactions for listeners and listener notifications are essentially the same as the interactions for connections and polling notifications, as described in "Connection Class Interactions" on page 62 and "Polling Notification Interactions" on page 112, respectively. This section describes the run-time behavior of listeners and their associated notifications.

Important:
A listener and the associated listener notifications must reside in the same package in the *Integration Server_directory* /packages directory. Otherwise, data can be lost when a package is reloaded.

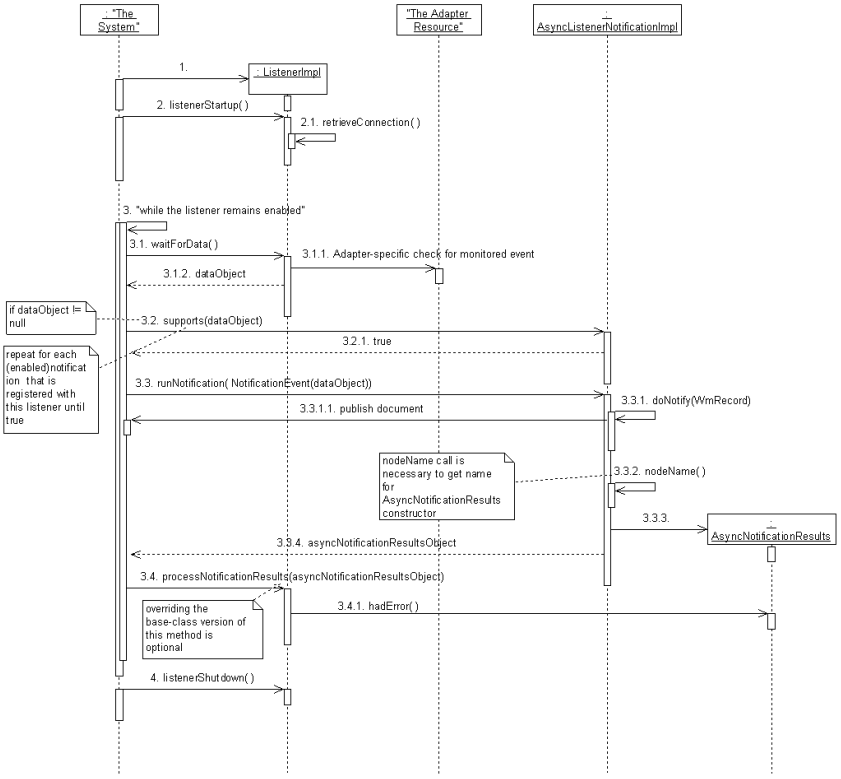
The figures that illustrate listener run-time interactions with synchronous and asynchronous notifications in this section show the run-time lifecycle of a listener from the time it is started until the time it is shut down. The difference between the two diagrams begins at step 3.3, when the server calls the runNotification method of the notification. Do not interpret the separation of these diagrams to imply that a given listener can use only synchronous or asynchronous notifications. On the contrary, step 3 represents a loop that repeats continuously while the listener is running,

and any iteration of the loop may follow either course depending on the class type of the notification that indicates support for the notification event.

When a listener starts (or restarts), the server retrieves a connection from the associated connection node before it calls the listener's listenerStartup method. This method can access the connection using the retrieveConnection method from the base class. The listenerStartup implementation should validate the values of any metadata parameter settings, and perform any initialization required prior to the first waitForData call. An exception thrown by the listenerStartup method, or a failure to retrieve a connection by the server, disables the listener.

Note:
A listener instance holds the connection it retrieves during listener initialization for the lifetime of the listener instance. Disabling the connection node will not impact this connection already held by the listener, but will prevent the listener from starting or restarting in the event of an AdapterConnectionException.

Listener run-time interactions with asynchronous notification



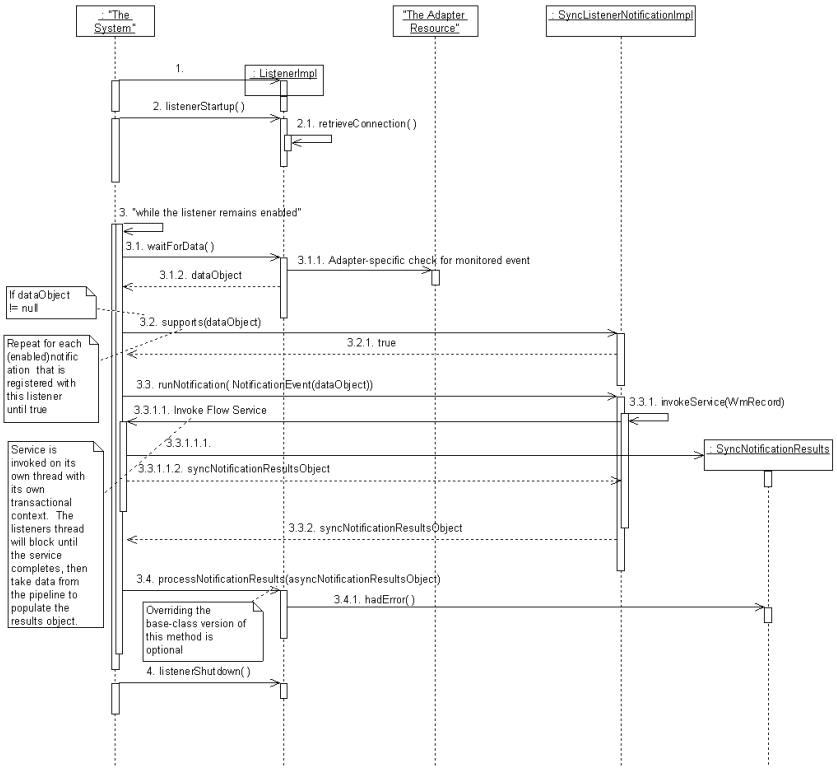
After initialization of the listener is complete, the server initiates the notification event-processing loop represented by step 3 of the interaction diagrams. This loop continues until one of the following events occurs:

- The adapter user disables the listener in the adapter's administrative interface.
- The package containing the listener is disabled (or reloaded).

Note:
Disable the listener package before you disable the adapter package.

- Integration Server is shut down.
- The listener or a listener notification throws an AdapterConnectionException. This causes the listener to shut down and to attempt to restart with a new connection.
- The listener throws an AdapterException or a RunTimeException.

Listener run-time interactions with synchronous notification



The event-processing loop begins with a call to the listener's waitForData method. This method should interrogate the adapter resource to determine whether an event has occurred that the listener should report. If not, the listener should return a null object to allow the server to check for the shutdown conditions mentioned above. The model assumes that the listener will implement some form of blocking read operation with a time component that will allow it to return periodically, even if no notification event has occurred. It is your responsibility to provide appropriate means of configuring the timing characteristics of this blocked read (such as through a metadata parameter on either the listener or the connection).

When the listener's waitForData method returns a non-null object, the server iterates through the listener notifications that are currently registered with the listener (which must be enabled and registered). For each notification, the supports method is passed the data object received from waitForData (step 3.2). (The notification list order can be manipulated in the adapter's administrative interface by editing the listener's settings.) For the first notification to return true from its supports method, the server calls the runNotification method, passing the data object wrapped as a

NotificationEvent (step 3.3). The NotificationResults object returned from runNotification is then passed to the listener's processNotificationResults method (step 3.4). If no notification returned true from the supports call, or if runNotification threw an exception (other than an AdapterConnectionException), then processNotificationResults is called with a null argument.

The listener notification's runNotification method (step 3.3) is responsible for interpreting the NotificationEvent object, and for building a WmRecord object consistent with its output signature. (This may or may not require additional interaction with the resource. If it does, a connection is made available for that purpose through the retrieveConnection method, which is the same connection used by the listener.) An asynchronous notification passes this WmRecord object to doNotify (step 3.3.1 in the figure that shows listener run-time interactions with asynchronous notification), which publishes the document in the same way it does for polling notifications. After publishing the notification, an asynchronous listener notification should instantiate a new AsynchNotificationResults object (step 3.3.3), including the notification name, which can be retrieved from the base class nodeName method (step 3.3.2).

The implementation of runNotification in a synchronous notification passes its output WmRecord object to invokeService instead of doNotify (step 3.3.1 in the figure that shows listener run-time interactions with synchronous notification). The service called by this method is specified in the configuration of the synchronous listener notification node. (As noted in the diagram, the service is invoked on a separate thread and transaction context.) The results of this service invocation are returned by invokeService as a SynchronousNotificationResults object. The results are then available to be interrogated by the notification and/or the listener, using the methods provided by that class.

Listener Implementation

The example listener implementation provided in this section shows the basic mechanics of a simple listener that can be used to monitor activity on an Integration Server log file. The example listener notification parses a session log entry and produces asynchronous notifications.

The tasks for implementing a listener are as follows:

- "Defining a WmConnectedListener Implementation Class" on page 138.
- "Updating the fillAdapterTypeInfo Method" on page 139.
- "Updating the Resource Bundle" on page 140.
- "Specifying Configuration Metadata" on page 140.
- "Implementing Run-Time Code" on page 141.
- "Configuring and Testing Listener Nodes and Listener Notification Nodes" on page 148.

Defining a WmConnectedListener Implementation Class

Create a class that extends com.wm.adk.notification.WmConnectedListener. The class must have at least skeletal implementation of the abstract methods. For example:

```
public class SimpleListener extends WmConnectedListener
{
    public void fillWmDescriptor(WmDescriptor descriptor, Locale locale)
```

```
        throws ResourceException
    {
    }
    public void listenerStartup() throws ResourceException
    {
    }
    public Object waitForData() throws ResourceException
    {
        return null;
    }
    public void listenerShutdown()
    {
    }
}
```

Implementing the shutdownCallBack() Method

Invoked on a thread separate from the listener's thread, this method allows the listener's waitForData loop to be gracefully interrupted prior to a normal shutdown. The thread that initiates listenerShutdown invokes shutdownCallBack in the following situations:

- When the listener node is disabled.
- When the package containing the listener node is reloaded.
- When Integration Server shuts down.

The system passes to this method a reference to the underlying resource connection. The method is invoked prior to calling the listener's listenerShutdown method. It will not be called if the shutdown is due to an exception.

The signature of this method is:

```
public void shutdownCallBack(WmManagedConnection wmConn) throws
                                                    ResourceException
```

Because shutdownCallBack is called from within an external thread (that is, not the same thread in which the listener itself is executing), you can take advantage of this feature to gracefully interrupt the normal functioning of the waitForData method. The listenerShutdown method will be subsequently invoked by WmART on the listener's thread. This is where you should perform any listener-specific cleanup tasks.

Updating the fillAdapterTypeInfo Method

Update the fillAdapterTypeInfo method of your main adapter class (the WmAdapter implementation) to register your listener notification type. For example:

```
public void fillAdapterTypeInfo(AdapterTypeInfo info, Locale locale)
{
    info.addListenerType(SimpleListener.class.getName());
...
```

Updating the Resource Bundle

Optionally, update the resource bundle with a display name, description, and help URL to make the listener more usable. For example:

```
,{SimpleListener.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
                                   "Simple Listener"}
,{SimpleListener.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_DESCRIPTION,
                                   "Use to monitor log files"}
,{SimpleListener.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_LISTLISTENERS
                                 + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL,
<help-URL>}
,{SimpleListener.class.getName() + ADKGLOBAL.RESOURCEBUNDLEKEY_LISTLISTENERTYPES
                                + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL, <help-URL>}
,{SimpleListener.class.getName() +
                                ADKGLOBAL.RESOURCEBUNDLEKEY_LISTLISTENERNOTIFICATIONS
                          + ADKGLOBAL.RESOURCEBUNDLEKEY_HELPURL, <help-URL>}
```

Specifying Configuration Metadata

The next logical step for implementing a listener is to create the metadata constructs that will enable adapter users to create listener notification nodes. You must create metadata parameters appropriate to the function of the listener, and describe the presentation and data entry rules for those parameters. This example only requires the name of the file that will be monitored. Create the metadata parameter by creating the accessor method and the associated fields as follows:

```
public static final String FILE_NAME_PARM = "fileName";
private String _fileName= null;
```

```
public void setFileName(String val){_fileName = val;}
```

Then update the descriptor:

```
public void fillWmDescriptor(WmDescriptor descriptor, Locale
locale)
    throws ResourceException
{
    descriptor.setRequired(FILE_NAME_PARM);
    descriptor.setDescriptions(
       MyAdapter.getInstance().getAdapterResourceBundleManager(),
       locale);
}
```

And finally, add the parameter to the resource bundle:

```
,{SimpleListener.FILE_NAME_PARM + ADKGLOBAL.RESOURCEBUNDLEKEY_DISPLAYNAME,
"Log File Name"}
```

Restricting Listeners to Register Specified Notification Templates

By default a listener supports all listener notification templates of the adapter. When adapter users configure a new listener notification (either synchronous or asynchronous) using Designer, they may select from the complete list of all enabled listeners in the adapter. You may want to override this default behavior so that the listener implementation specifies exactly which notification

templates it supports. To do this, implement the method restrictNotificationTypes in your listener implementation class. This method returns a String array containing the fully qualified path names of one or more notification template classes that the listener will support. Then, when adapter users attempt to configure a new listener notification in Designer, only those template classes in the returned array will appear in the Adapter Notification Editor. The following example shows how to use this method.

```
public class SimpleListener extends WmConnectedListener
{
...
   public String[] restrictNotificationTypes()
   {
      return new String[]{"com.wm.myadapter.notifications.FooNotification",
                          "com.wm.myadapter.notifications.BarNotification"};
   }
...
}
```

Note:
All names returned by this method should refer to classes that extend WmNotification.

Implementing Run-Time Code

The run-time code for the example listener manages a FileReader object and variables to hold data read from the log as follows:

```
private FileReader _reader = null;
private StringBuffer workingBuffer = new StringBuffer();
private String _lastDataObject = null;
```

```
public void listenerStartup() throws ResourceException
{
    try
    {
        _reader = retrieveConnection().getReader(_fileName);
        while(_reader.ready())
        {
           _reader.read();  // move to the end of the stream
        }
    }
    catch (Throwable t)
    {
        throw MyAdapter.getInstance().createAdapterException(100,t);
    }
}
public Object waitForData() throws ResourceException
{
    try
    {
        if(_reader.ready())
        {
           do
           {
               int i = _reader.read();
               if (i != -1)
               {
```

```
                    char c = (char)i;
                    workingBuffer.append(c);
                    if(c == '\n')
                    {
                       _lastDataObject = new String(workingBuffer);
                       workingBuffer = new StringBuffer();
                       break;
                    }
               }
               else
               {
                    break;
               }
          }
          while (_reader.ready());
     }
}
     catch (Throwable t)
     {
          throw MyAdapter.getInstance().createAdapterException(100,t);
     }
     return _lastDataObject;
}
public void listenerShutdown()
{
     try
     {
          _reader.close();
     }
     catch(Throwable t){}
}
```

The example also includes an implementation of processNotificationResults:

```
public void processNotificationResults(NotificationResults results)
     throws ResourceException
{
     if(results != null)
     {
          if(results.hadError())
          {
              MyAdapter.getLogger().logError(9999,
              "Error processing: " + this._lastDataObject
              +" errorInfo = " + results.getErrorInfo().toString());
          }
     }
          else
     {
     MyAdapter.getLogger().logError(
     9999,"No notification available to process:" + this._lastDataObject);
     }
}
```

Listener Notification Implementation

The following example implements an asynchronous listener notification that recognizes session log entries and generates notifications from them. To distinguish a session log entry from another type of log entry, the listener parses the entry data in the notification's supports method. The

supports method model is flexible enough to permit this approach because the same listener notification object instance that returns true from the supports call is guaranteed to receive the runNotification call.

The example listener notification uses an alternative approach to implementing resource domains that redirects resource domain activities back to the notification (or adapter service) that uses it. This model does a better job of encapsulating the notification functionality into a single class. The model is described in detail in "An Alternative Approach to Organizing Resource Domains" on page 176. If you do not want to use this model, you may implement the resource domain code following the model described in "Specifying Adapter Service Signature Resource Domains" on page 99.

Another new concept used in this example is the "uses" metadata mechanism. This is a shortcut mechanism commonly used to manipulate a metadata signature. You use it to add a column of check boxes to the adapter's interface so that adapter users can select the fields to use for the notification. When they select the check boxes, values will appear in the Signature column. This metadata mechanism is described in "The useParam Argument of setResourceDomain" on page 144, but you do not need to fully understand the constraint of that feature to implement this example.

The tasks for implementing an asynchronous listener notification are as follows:

- "Defining a WmAsyncListenerNotification Implementation Class" on page 143.
- "Updating the fillAdapterTypeInfo Method" on page 144.
- "Specifying Metadata" on page 144.
- "Implementing Resource Domains" on page 146.
- "Implementing the supports and runNotification Methods" on page 147.
- "Configuring and Testing Listener Nodes and Listener Notification Nodes" on page 148.

Defining a WmAsyncListenerNotification Implementation Class

Create a class that extends com.wm.adk.notification.WmAsyncListenerNotification. This class should provide default implementations for the abstract methods fillWmTemplateDescriptor, supports, and runNotification. Because the example uses the ResourceDomainHandler interface, it is also necessary to provide default implementations of the methods defined in that interface. (For information about the ResourceDomainHandler interface, see "An Alternative Approach to Organizing Resource Domains" on page 176.) If you are not using the alternative resource domain management model, skip the first three methods in the following listing:

```
public class SessionLogListenerNotification extends WmAsyncListenerNotification
implements ResourceDomainHandler
{
```

```
    public Boolean adapterCheckValue(
       WmManagedConnection connection,
       String resourceDomainName,
       String[][] values,String testValue) throws AdapterException
    {
       return null;
    }
```

```
    public ResourceDomainValues[] adapterResourceDomainLookup(
       WmManagedConnection connection,
       String resourceDomainName,
       String[][] values) throws AdapterException
    {
        return null;
    }
    public void registerResourceDomain(
        WmManagedConnection connection,
        WmAdapterAccess access) throws AdapterException
    {
    }
    public void fillWmTemplateDescriptor(WmTemplateDescriptor
                                         descriptor,Locale l)
       throws ResourceException
    {
    }
    public boolean supports(Object data) throws ResourceException
    {
        boolean result = false;
        return result;
    }
```

```
    public NotificationResults runNotification(NotificationEvent event)
        throws ResourceException
    {
        NotificationResults result = null;
        return result;
    }
}
```

Updating the fillAdapterTypeInfo Method

Next, register the new notification type in the fillAdapterTypeInfo method of the main adapter class, as follows:

```
    public void fillAdapterTypeInfo(AdapterTypeInfo info, Locale locale)
    {
    ...
info.addNotificationType(SessionLogListenerNotification.class.getName());
    ...
```

Specifying Metadata

The example listener notification in this section includes only those configuration parameters that are directly related to the signature. The only new concept in this section is the use of the useParam argument of the WmTemplateDescriptor.setResourceDomain method.

The useParam Argument of setResourceDomain

The WmTemplateDescriptor interface provides two signatures for the setResourceDomain method, one of which includes the argument useParam:

```
public void setResourceDomain(java.lang.String name,
                              java.lang.String resourceDomainName,
```

```
                                java.lang.String[] dependencies,
                                java.lang.String useParam)
```

You use this argument as a filter to determine which of the available fields will be included in the signature. The example in this section implements the useParam argument as USES_PARM, in the setResourceDomain method near the end of the example.

This parameter is only meaningful when the parameter identified in the name argument and the parameter names in useParam are in the same fieldMap, and the data type of the useParam parameter is boolean[]. When these conditions are satisfied, the Adapter Service Editor will treat the useParam setting as a filter on the resource domain association. When this occurs, the resource domain values will only be applied to the name parameter row in the fieldMap if the corresponding value in the useParam is true (that is, when the adapter user selects the check box).

When the adapter user selects the fields to use for the notification (by selecting the check boxes in the Uses column), values appear in the Signature column. When you save changes to the notification node, the signature changes are reflected in the associated document type.

```
public static final String FIELD_NAMES_PARM = "fieldNames";
public static final String FIELD_TYPES_PARM = "fieldTypes";
public static final String USES_PARM = "uses";
public static final String SIG_PARM = "signature";
public static final String NOTIFICATION_SETUP_GROUP =
   "SessionLogListenerNotification.setup";
public static final String FIELD_NAMES_RD =
   "SessionLogListenerNotification.fieldNames.rd";
public static final String FIELD_TYPES_RD =
   "SessionLogListenerNotification.fieldTypes.rd";
private String[] _fieldNames = null;
private String[] _fieldTypes = null;
private boolean[] _uses = null;
public void setFieldNames(String[] val){_fieldNames = val;}
public void setFieldTypes(String[] val){_fieldTypes = val;}
public void setUses (boolean[] val){_uses = val;}
public void setSignature(String[] val){}
public void fillWmTemplateDescriptor(WmTemplateDescriptor descriptor,Locale l)
   throws ResourceException
   {
   String[] parms = new String[] {FIELD_NAMES_PARM,
                                  FIELD_TYPES_PARM,
                                  USES_PARM,
                                  SIG_PARM};
   descriptor.createGroup(NOTIFICATION_SETUP_GROUP,parms);
   descriptor.createFieldMap(parms,false);

   descriptor.createTuple(new String[]{FIELD_NAMES_PARM,
                                       FIELD_TYPES_PARM});
   descriptor.setResourceDomain(FIELD_NAMES_PARM,FIELD_NAMES_RD,null);
   descriptor.setResourceDomain(FIELD_TYPES_PARM,FIELD_TYPES_RD,null);
   descriptor.setResourceDomain(SIG_PARM,
      WmTemplateDescriptor.OUTPUT_FIELD_NAMES,
      new String[]{FIELD_NAMES_PARM,FIELD_TYPES_PARM}, USES_PARM);
   descriptor.setDescriptions(
      MyAdapter.getInstance().getAdapterResourceBundleManager(),l);
}
```

Implementing Resource Domains

As previously mentioned, this example uses an alternative approach to managing its resource domains. This is evident in the modifications to the signatures of the three resource domain value methods shown below. Completing this implementation also requires some changes to the way your connection is implemented, as described in "An Alternative Approach to Organizing Resource Domains" on page 176. Alternatively, you may implement the registerResourceDomain method shown below in your connection factory, and the remaining methods in your connection, as shown in previous sections of this document.

Note:
In this example, notice the use of `class.getName` when specifying data types. Using this technique enables you to catch spelling errors at compile time.

```
private static final String[] _sigFieldNames ={
    "timeStamp",
    "component",
    "rootContext",
    "parentContext",
    "currentContext",
    "server",
    "eventCode",
    "user",
    "sessionName",
    "RPCs",
    "age"};
public Boolean adapterCheckValue(
    WmManagedConnection connection,
    String resourceDomainName,
    String[][] values,String testValue) throws AdapterException
    {
    return null;
    }
public ResourceDomainValues[] adapterResourceDomainLookup(
    WmManagedConnection connection,
    String resourceDomainName,
    String[][] values) throws AdapterException
{
    ResourceDomainValues[] results = null;
    if (resourceDomainName.equals(FIELD_NAMES_RD)
       || resourceDomainName.equals(FIELD_TYPES_RD))
    {
       ResourceDomainValues names =
          new ResourceDomainValues(FIELD_NAMES_RD,_sigFieldNames);
       ResourceDomainValues types =
          new ResourceDomainValues(FIELD_TYPES_RD,new String[] {
          Date.class.getName(),       //timestamp
          String.class.getName(),     // component
          String.class.getName(),     // rootContext
          String.class.getName(),     // parentContext
          String.class.getName(),     // currentContext
          String.class.getName(),     // server
          Integer.class.getName(),    // eventCode
          String.class.getName(),     // user
          String.class.getName(),     // sessionName
          Integer.class.getName(),    // RPCs
```

```
          Long.class.getName()        // age
          });
       results = new ResourceDomainValues[] {names,types};
         }
         return results;
}
public void registerResourceDomain(WmManagedConnection connection,
   WmAdapterAccess access)
      throws AdapterException
{
      access.addResourceDomainLookup(
         this.getClass().getName(),FIELD_NAMES_RD,connection);
      access.addResourceDomainLookup(
         this.getClass().getName(),FIELD_TYPES_RD,connection);
}
```

Implementing the supports and runNotification Methods

This example implementation of the supports method parses the contents of the data object (originally returned from the listener's waitForData method) into the object variable _parsedValues. Any error in the parsing process indicates that the data object was not a session log entry, and the supports method returns a false value.

```
private Object[] _parsedValues = new Object[_sigFieldNames.length];
public boolean supports(Object data) throws ResourceException
{
   boolean result = false;
      try
      {                                        //2003-08-09 13:05:20 EDT
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd H:mm:ss zzz");
        String sData = (String)data;
        this._parsedValues[0] = sdf.parse(sData.substring(0,48));
         StringTokenizer st = new StringTokenizer(sData.substring(49)," ",false);
        this._parsedValues[1] = st.nextToken();
        this._parsedValues[2] = st.nextToken();
        this._parsedValues[3] = st.nextToken();
        this._parsedValues[4] = st.nextToken();
        st.nextToken();     // skip the session ID
        this._parsedValues[5] = st.nextToken();
        this._parsedValues[6] = new Integer(st.nextToken());
        this._parsedValues[7] = st.nextToken();
        this._parsedValues[8] = st.nextToken();
        this._parsedValues[9] = new Integer(st.nextToken());
        this._parsedValues[10] = new Long(st.nextToken());
        result = true;
     }
     catch(Throwable t){}
  return result;
}
```

The runNotification implementation assumes that the supports method was successful in parsing the data object, so it does not need the NotificationEvent argument. It merely populates a WmRecord object by inserting the parsed names, using the same key names array that populate the resource domain.

```
public NotificationResults runNotification(NotificationEvent event)
   throws ResourceException
```

```
{
   NotificationResults result = null;
   WmRecord notice = WmRecordFactory.getFactory().createWmRecord("notUsed");
   for(int i = 0; i< _sigFieldNames.length;i++)
   {
       if (_uses[i])
       {
          notice.put(_sigFieldNames[i],_parsedValues[i]);
       }
   }
   this.doNotify(notice);
   result = new AsyncNotificationResults(this.nodeName(),true,null);
   return result;
}
```

Note:
The doNotify method has two forms. When calling the WmAsynchronousNotification.doNotify(WmRecord rec, String msgId) form of this method, the length (the number of characters) of the value in msgId should not exceed a particular limit. To determine this limit, call the WmAsynchronousNotification.adapterMaxMessageIdLen() method. There is a fixed number of characters available in Integration Server to hold a notification ID. Of these, the WmART package reserves a certain number to hold a unique ID that it inserts prior to dispatching a notification. The remaining characters are available to you when calling WmAsynchronousNotification.doNotify(WmRecord rec, String msgId). For more information, see the Javadoc for WmAsynchronousNotifcation.doNotify.

Configuring and Testing Listener Nodes and Listener Notification Nodes

Now you are ready to configure a listener node and a listener notification node as follows:

- Compile your adapter as described in "Compiling the Adapter" on page 43.
- Reload your adapter as described in "Loading, Reloading, and Unloading Packages" on page 152.
- Configure and enable a listener node as described in "Configuring Listener Nodes" on page 160.
- Configure and enable a listener notification node as described in "Configuring the checkDepositListener Notification Node" on page 268.

8 Design-Time Tasks

- Overview 150
- Package Management 150
- Configuring Connection Nodes 153
- Configuring Adapter Service Nodes 156
- Polling Notification Nodes 157
- Listener Notification Nodes 159

Overview

At design time, adapter users will configure and initialize the following run-time components of the adapter:

- **Namespace node packages**. Management tasks include:
  - Setting package dependencies for namespace node packages (see "Package Dependency Considerations for Namespace Node Packages" on page 151).
  - Setting package dependencies for namespace nodes (see "Package Dependency Considerations for Namespace Nodes" on page 151).
  - Using access control lists (ACLs) to control which development group has access to which adapter services (see "Group Access Control" on page 152).
  - "Enabling and Disabling Packages" on page 152).
  - "Loading, Reloading, and Unloading Packages" on page 152).
- **Connection nodes** (see "Configuring Connection Nodes" on page 153).
- **Adapter service nodes** (see "Configuring Adapter Service Nodes" on page 156).
- **Polling notification nodes** (see "Polling Notification Nodes" on page 157).
- **Listener nodes and listener notification nodes** (see "Listener Notification Nodes" on page 159).

To perform these tasks, adapter users will use webMethods Integration Server, Software AG Designer, and a web browser.

Note:
You must have Integration Server administrator privileges to access an adapter's management screen. For information about setting user privileges, see the *webMethods Integration Server Administrator's Guide* for your release.

Package Management

A namespace node package is a package in which an adapter user will create the adapter's namespace nodes for connections, adapter services, polling notifications, listener notifications, and listeners. They should not create nodes in the adapter package. Keeping namespace nodes separate from the adapter package simplifies the process of upgrading a deployed adapter.

The procedure for creating a namespace node package is identical to the procedure for creating any webMethods package. For instructions for creating packages, see the *webMethods Service Development Help* for your release.

All the nodes of an adapter may be located in one package, or they may be distributed among multiple packages.

Package Dependency Considerations for Namespace Node Packages

Following are dependency requirements and guidelines for namespace node packages:

- A namespace node package must have a dependency on its associated adapter package, and the adapter package must have a dependency on the WmART package. For more information about setting package dependencies, see the *webMethods Service Development Help* for your release.

  Setting these dependencies ensures that at startup Integration Server automatically loads or reloads all packages in the proper order: the WmART package first, the adapter package next, and the node package(s) last. The WmART package is automatically installed when you install Integration Server. You should not need to manually reload the WmART package.

- Keep connections for different adapters in separate packages so that you do not create interdependencies between adapters. If a package contains connections for two different adapters, and you reload one of the adapter packages, the connections for both adapters will reload automatically.

- Integration Server will not allow you to enable a package if it has a dependency on another package that is disabled. That is, before you can enable your package, you must enable all packages on which your package depends. For information about enabling packages, see "Enabling and Disabling Packages" on page 152.

- Integration Server will allow you to disable a package even if another package that is enabled has a dependency on it. Therefore, you must manually disable any user-defined packages that have a dependency on your adapter package before you disable the adapter package.

If the namespace nodes of an adapter are located in multiple packages, see "Package Dependency Considerations for Namespace Nodes" on page 151. For more information about setting package dependencies, see the *webMethods Service Development Help* for your release.

Package Dependency Considerations for Namespace Nodes

If all the namespace nodes of an adapter are located in the same package, there is no need to set package dependencies.

However, if the nodes of an adapter are located in multiple packages, then:

- A package that contains the connection node(s) must depend on the adapter package.

- Packages that contain other nodes must depend on their associated connection package.

For more information about setting package dependencies, see the *webMethods Service Development Help* for your release.

Group Access Control

To control which development group has access to which adapter services, use access control lists (ACLs). You can use ACLs to prevent one development group from inadvertently updating the work of another group, or to allow or deny access to services that are restricted to one group but not to others.

For general information about assigning and managing ACLs, see the *webMethods Service Development Help* for your release.

Enabling and Disabling Packages

All packages are automatically enabled by default. To prevent Integration Server from loading a particular package, you must manually disable that package, using the Management screen of Integration Server Administrator. To do this, click **Yes** in the Enabled column for the package; the Yes changes to No (disabled). Disabled packages are not listed in Designer. A disabled adapter will remain disabled until you explicitly enable it using Integration Server Administrator.

To re-enable a package, on the Integration Server Administrator > Packages > Management screen, click **No** in the **Enabled** column.

The **Enabled** column changes its value to **Yes** (enabled).

Note:
Enabling an adapter package will not cause its associated node package(s) to be reloaded. For information about reloading packages, see "Loading, Reloading, and Unloading Packages" on page 152.

Important:
Before you manually enable a node package, *you must first enable its associated adapter package*. Similarly, if your adapter has multiple node packages, and you want to disable some of them, disable the adapter package first. Otherwise, errors will be issued when you try to access the remaining enabled node packages.

Loading, Reloading, and Unloading Packages

If the node packages are properly configured with a dependency on the adapter package (as described in "Package Dependency Considerations for Namespace Node Packages" on page 151), at startup Integration Server automatically loads or reloads all packages in the proper order: the WmART package first, the adapter package next, and the node package(s) last. You should not need to manually reload the WmART package.

Reloading Packages Manually

You will typically reload the adapter package manually as you make changes to the adapter code. Similarly, adapter users will reload their node packages manually when they modify the nodes. Reloading a node package will not cause its associated adapter package to be reloaded.

You can reload adapter packages and node packages from either Integration Server Administrator (by clicking the reload icon in the **Reload** column of the Management screen) or from Designer (by right-clicking the package and selecting the Reload Package option from the menu).

Unloading Packages

At shutdown, Integration Server unloads the packages in the reverse order in which it loaded them: it unloads the node package(s) first, the adapter package next, and the WmART package last (assuming the dependencies are correct).

Configuring Connection Nodes

You create, manage, and enable connection namespace nodes using the adapter's management screen. When you create a connection node, it is disabled by default. You must enable the connection node after you create it.

To configure a connection node

1. On the Integration Server Administrator > Adapters screen, select your adapter.

   An adapter management screen opens, displaying any connection nodes that are currently configured for the adapter. There may be one connection type for each connection factory supported by the adapter.

2. Select **Connections**from the navigation area.

3. Click **Configure New Connection**.

4. On the Connection Types screen, select a connection type.

5. On the Configure Connection Type screen, provide values for the connection's parameters.

   a. Complete the Configure Connection Type > *adapter_name* section as follows:

Field	Description
Package	Select a namespace node package in which to create the connection. For more information about creating packages, see "Package Management" on page 150.
Folder Name	Type the folder name in which to create the connection. If the folder does not already exist in the package, the server creates it.
Connection Name	Type a connection name.

   b. Complete the Connection Properties section as appropriate for your adapter resource.

Enter values for the connection's metadata properties. For example, the Sample Adapter displays parameters such as Sample Server Host, Sample Server Port Number, Local Transaction Control, and Timeout.

c. Complete the Connection Management Properties section as follows:

Field	Description
Enable Connection Pooling	Enables or disables the use of connection pooling for the connection. The default value is `true` (enabled).
Minimum Pool Size	The number of connections to create when the connection is enabled. The default value is `1`.
Maximum Pool Size	The maximum number of connections that can exist at one time in the connection pool. The default value is `10`.
Pool Increment Size	The number of connections by which the pool will be incremented if connections are needed, up to the maximum pool size. The default value is `1`.
Block Timeout	If connection pooling is enabled, this field specifies the number of milliseconds that Integration Server will wait to obtain a connection with the adapter resource before it times out and returns an error. For example, you have a pool with **Maximum Pool Size** of 20. If you receive 30 simultaneous requests for a connection, 10 requests will be waiting for a connection from the pool. If you set the **Block Timeout** to 5000, the 10 requests will wait for a connection for 5 seconds before they time out and return an error. If the services using the connections require 10 seconds to complete and return connections to the pool, the pending requests will fail and return an error message stating that no connections are available. If you set the **Block Timeout** value too high, you may encounter problems during error conditions. If a request contains errors that delay the response, other requests will not be sent. This setting should be tuned in conjunction with the **Maximum Pool Size** to accommodate such bursts in processing. The default value is `1000`.
Expire Timeout	If connection pooling is enabled, this field specifies the number of milliseconds that an inactive connection can remain in the pool before it is closed and removed from the pool. The connection pool will remove inactive connections until the number of connections in the pool is equal to the **Minimum Pool Size**. The inactivity timer for a connection is reset when the connection is used by the adapter. If you set the **Expire Timeout**value too high, you may have a number of unused inactive connections in the pool. This consumes

Field	Description
	local memory and a connection on your backend resource. This could have an adverse effect if your resource has a limited number of connections. If you set the **Expire Timeout**value too low, performance could degrade because of the increased activity of creating and closing connections. This setting should be tuned in conjunction with the **Minimum Pool Size** to avoid excessive opening/closing of connections during normal processing. The default value is `1000`. Enter `-1` to specify no timeout.
Startup Retry Count	The number of times that the system should attempt to initialize the connection pool at startup if the initial attempt fails. The default value is `0` (a single attempt).
Startup Backoff Timeout	The number of seconds that the system should wait between attempts to initialize the connection pool. This field is irrelevant if the value of **Startup Retry Count** is `0`. The default value is `10`.

6. Click **Test Connection**.

   The connection is tested based on the settings provided.

7. Click **Save Connection**.

   The connection name is now listed on the adapter's Connections screen and in the Service Browser of Designer.

8. On the adapter's Connections screen, enable the connection node by clicking **No** in the **Enabled** column. The **Enabled** column now shows **Yes**.

   The server initializes a connection pool based on the provided settings. Enabling and disabling a connection node produces entries similar to these in the server log:

```
2003-08-18 08:21:36 EDT [ART.0118.5505V1] Adapter Runtime (Connection):
  Starting connection connections:sampleConnection.
2003-08-18 08:21:36 EDT [ART.0118.5517V1] Adapter Runtime (Connection):
  Creating connection manager properties:>>>BasicData:poolable=true,
  minimumPoolSize=1,maximumPoolSize=10,poolIncrementSize=1,
  blockingTimeout=20000,expireTimeout=20000,selectionSize=1<<<.
2003-08-18 08:21:36 EDT [ADA.0502.0101D] Initializing Sample Connection
2003-08-18 08:21:36 EDT [SCC.0126.0001E] SCC connectionManager Pool Started
2003-08-18 08:24:10 EDT [ART.0118.5510V1] Adapter Runtime (Connection):
  Stopping connection connections:sampleConnection.
```

Note:
If a connection node is enabled when the server shuts down, it will be enabled at server startup.

Configuring Adapter Service Nodes

Before you configure adapter service nodes, ensure that you have configured a connection node as described in "Configuring Connection Nodes" on page 153. You can use Designer to configure adapter service nodes.

To configure adapter service nodes

1. Start Designer.

   Note:
   Make sure that the Integration Server, with which you want to use Designer, is running.

2. Select a namespace node package in which to create the adapter service node.

3. Create a folder in the selected package and navigate to that folder in the Package Navigator section of Designer.

4. Select **File** > **New**.

5. Select **Adapter Service** from the list of elements.

6. On the Create a New Adapter Service screen, type a name for your service in the **Element name** field and click **Next**.

7. On the Select Adapter Type screen, select the name of your adapter and click **Next**.

8. On the Select an Adapter Connection Alias screen, select the appropriate adapter connection name and click **Next**.

   Important:
   If you are using the adapter with Integration Server 8.0 SP1 or earlier, adapter services cannot use connections that are also used for adapter listeners.

9. On the Select a Template screen, select an adapter service template and click **Next**.

10. Click **Finish**.

11. Specify values for the tab that is specific for your adapter resource (such as Query, Update, Add, or Delete tab).

12. Specify values for the Input/Output tab and the Settings tab. For more information, see the *webMethods Service Development Help* for your release.

13. Select **File** > **Save**.

Polling Notification Nodes

Before you configure a polling notification node, ensure that you have configured a connection node as described in "Configuring Connection Nodes" on page 153. This section provides procedures for:

- "Configuring Polling Notification Nodes" on page 157.
- "Scheduling and Enabling Polling Notification Nodes" on page 158.

Configuring Polling Notification Nodes

Perform the following procedure to configure a polling notification node. You can use Designer to configure a polling notification node.

To configure a polling notification node

1. Start Designer.

   Note:
   Make sure that the Integration Server, with which you want to use Designer, is running.

2. Select a namespace node package in which to create the adapter service node.

3. Create a folder in the selected package and navigate to that folder in the Package Navigator section of Designer.

4. Select **File** > **New**.

5. Select **Adapter Notification** from the list of elements.

6. On the Create a New Adapter Notification screen, type a name for your service in the **Element name** field and click **Next**.

7. On the Select Adapter Type screen, select the name of your adapter and click **Next**.

8. On the Select a Template screen, select an adapter notification template and click **Next**.

9. On the Select an Adapter Connection Alias screen, select the appropriate adapter connection name and click **Next**.

   Important:
   If you are using the adapter with Integration Server 8.0 SP1 or earlier, adapter services cannot use connections that are also used for adapter listeners.

10. Click **Finish**.

11. Select **File** > **Save**.

    The adapter creates the notification node and a document (named *notificationName*PublishDocument). This document is used to contain the data of the affected portion of the adapter resource (such as a database row), and to inform Integration Server of the changes.

12. In the Adapter Notification Editor's **Message Polling** tab, select a polling event from the **Polling Name** field.

13. Configure the Adapter Notification Editor fields as appropriate for your adapter resource. For information about using the Adapter Notification Editor, see the *webMethods Service Development Help* for your release.

14. Select **File** > **Save**.

15. Schedule and enable the notification at runtime, as described in "Scheduling and Enabling Polling Notification Nodes" on page 158.

Scheduling and Enabling Polling Notification Nodes

Before you can use a notification, you must schedule and enable it.

To schedule and enable a polling notification node

1. Start Integration Server Administrator.

2. On the Integration Server Administrator > Adapters screen, select your adapter.

3. Select **Polling Notifications**.

4. On the **Polling Notifications** screen, use the following options to schedule and enable each polling notification:

   Note:
   If you use an XA-Transaction connection, you cannot enable a notification.

Option	Description/Action
Notification Name	The name of the notification.
Package Name	The name of the package for the notification.
State	**Note:** Before you can enable a polling notification, you must schedule it, using the **Edit Schedule** icon described in this procedure.

Option	Description/Action
	Once you schedule a polling notification, you can use this option to enable (**Yes**) or disable (**No**) a polling notification. Click on the current value in this field to change its value. If there is no polling notification scheduled for a given notification, **Not Scheduled**appears in this field. Use the use the **Edit Schedule** icon to create a polling notification as described in step 5.
Edit Schedule	Click on the **Edit Schedule** icon to create or modify polling notification parameters. **Note:** You must disable a polling notification before you can edit it.
View Schedule	Click on the **View Schedule** icon to review the parameters for the selected polling notification. Click **Return to Notifications** to go back to the main polling notification page.

5. To create or modify schedule parameters for the selected notification, click the **Edit Schedule** icon and set the following options:

Option	Description/Action
Interval (seconds)	Type the polling interval time in seconds.
Overlap	This option determines when the scheduled interval time you set in the **Interval** field begins. Enable this option to allow for executions of the scheduled notification to overlap. With the Overlap option enabled, the next scheduled execution does not wait for the current execution to end. If your notification requires the preservation of the notification ordering, do not enable this option.
Immediate	Enable this option to start polling immediately.

6. Click **Save Schedule**.

Note:
One way to test a polling notification node is presented in "Testing the underBalancePolling Notification Node" on page 262.

Listener Notification Nodes

Before you begin, ensure that you have configured a connection node as described in "Configuring Connection Nodes" on page 153. This section provides procedures for:

- "Configuring Listener Nodes" on page 160.
- "Configuring the checkDepositListener Notification Node" on page 268.
- "Enabling Listener Notification Nodes and Listener Nodes" on page 162.

Configuring Listener Nodes

Perform the following procedure to configure a listener node. You can use Designer to configure adapter service nodes.

To configure a listener node

1. On the Integration Server Administrator > Adapters screen, select your adapter.
2. Click **Listeners**.
3. On the Listener screen, click **Configure New Listener**.
4. On the Listener Types screen, select the appropriate listener type defined for the adapter.
5. Complete the Configure Listener Type > a*dapter_name* section as follows:

Field	Description\Action
Package	Select a namespace node package in which to create the listener.
Folder Name	Assign a name to the folder in which you will create the listener.
Listener Name	Assign a name to the listener node.
Connection Name	Select a connection node. **Important:** If you are using the adapter with Integration Server 8.0 SP1 or earlier, you must use separate connections for listeners and adapter services as well as listeners and polling notifications.
Retry Limit	The number of times that the system should attempt to start the listener if the initial attempt fails. Specifically, it specifies how many times to retry the listenerStartup method before issuing an AdapterConnectionException. When the value is set to `0`, the system makes a single attempt. The default value is 5.
Retry Backoff Timeout	The number of seconds the system should wait between each attempt to start the listener. This field is irrelevant if the value of **Retry Limit** is `0`. The default value is `10`.

6. Click **Save Listener**.

Note:
Enabling the listener before you configure and enable its corresponding listener notification node produces a warning.

Note:
A listener instance holds the connection it retrieves during listener initialization for the lifetime of the listener instance. Disabling the connection node will not impact the connection already held by the listener, but it will prevent the listener from starting or restarting in the event of an AdapterConnectionException.

Configuring Listener Notification Nodes

A listener notification node can be either synchronous or asynchronous. When an event occurs, an asynchronous listener notification publishes a document. You need to create a trigger that receives the document and executes a service to process the document's data. With a synchronous listener notification, you designate a flow service to process the data produced by the notification. A synchronous notification does not use a trigger.

To configure a listener notification node

1. Start Designer.
2. Select a namespace node package in which to create the adapter service node.
3. Create a folder in the selected package.
4. Select **File** > **New**.
5. Select **Adapter Notification** from the list of elements.
6. On the Create a New Adapter Notification screen, type a name for your service in the **Element name** field and click **Next**.
7. On the Select Adapter Type screen, select the name of your adapter and click **Next**.
8. On the Select a Template screen, select a listener notification template and click **Next**.
9. On the Select an Adapter Connection Alias screen, select the appropriate adapter connection name and click **Next**.
10. Perform one of the following steps:
    - If you selected an asynchronous listener notification template, click **Finish**.
    - If you selected a synchronous listener notification template, select a flow service to process the data produced by the notification, click **Next**, and then click **Finish**.

The Adapter Notification Editor opens.

11. Select an event from the **Event Name** field.

12. Select **File** > **Save**.

    If you selected an asynchronous listener notification template, the adapter creates a listener notification and a document (named *notificationName*PublishDocument). The document will be used to contain the data of the affected portion of the adapter resource (such as a database row), and to inform Integration Server of the changes. (Documents are not created for synchronous listener notifications.)

    When choosing the service for a synchronous notification, ensure that it does not require session data. For more information, see "Synchronous and Asynchronous Listener Notifications" on page 130.

Enabling Listener Notification Nodes and Listener Nodes

Enabling the listener before you configure and enable its corresponding listener notification node produces a warning.

To enable a listener node and a listener notification node

1. On the Integration Server Administrator > Adapters screen, select your adapter.

2. Click **Listener Notifications**.

3. On the Listener Notification screen, enable the listener notification node by clicking **No** in the **Enabled** column. The **Enabled** column now shows **Yes** (enabled).

4. On your adapter management screen, click **Listeners**.

5. On the Listeners screen, enable the listener by clicking **No** in the **Enabled** column. The **Enabled** column now shows **Yes** (enabled).

Note:
One way to test a listener notification node is presented in "Testing the checkDepositListener Notification Node" on page 271.

9 Run Time Activities

- Overview 164
- Retry and Recovery Architecture 164
- Run Time Connection Allocation for Adapter Services 167

Overview

A well designed adapter should include the following run-time capabilities:

- Ability to identify, and recover from, temporary errors (see "Retry and Recovery Architecture" on page 164 below).
- Ability to retrieve and manage connections, and allow the user to dynamically control the type of connection used for each service invocation (see "Run Time Connection Allocation for Adapter Services" on page 167).

Retry and Recovery Architecture

A highly available and reliable system has the ability to recover from temporary errors. If, during execution, a transient error is encountered, the system should do the following:

1. Detect the error
2. Remove or regenerate the component causing the error
3. Retry the operation.

The first phase in recovery is defining a transient error. For a webMethods adapter, a transient error is one that will go away in time, without requiring human intervention on webMethods Integration Server. If an adapter service is called and tries to perform an insert and the backend resource rejects the insert because the data format is incorrect, then this is not transient. Someone will need to reformat the data for the insert to complete correctly.

Suppose you try the insert and your backend system is offline for scheduled maintenance. The insert will fail, but when the system comes back on line, the insert would work. Retrying the operation in this scenario is useful.

The role of detecting the transient error is the responsibility of the adapter. The removal, regeneration, and retry functions are provided by Integration Server.

Detection

The ability to recognize these errors is critical in every component of an adapter. It is up to the adapter developer to determine which backend errors are transient. When an adapter recognizes this situation, it throws an AdapterConnectionException to alert Integration Server to the error. This exception is a subclass of the ResourceException. This is a valid exception to throw from any ADK method that declares, "throws ResourceException".

An adapter writer should make an effort to isolate the errors that are transient. When Integration Server catches the AdapterConnectionException it will initiate the retry operations. If this occurs for non-transient errors it will be a waste of processing time.

Removal/Regeneration

Transient exceptions within adapters are all related to connection errors. Therefore, the connection used in the operation will need to be cleared and regenerated before a retry occurs.

Based on the AdapterConnectionException received from the adapter, Integration Server will clean the entire pool or just the current connection. The AdapterConnectionException will indicate if the entire pool needs to be cleaned or only the current connection. By default, the exception will indicate the entire pool needs to be cleaned. For most applications this is the correct behavior.

To clean the current connection, the destroy() method is called and the object is discarded. To clean the pool, the connection pool manager will be notified and all free connections will be destroyed and discarded. Any busy connections will be destroyed and discarded when they are released. The connection node will remain enabled.

The regeneration occurs when a new connection is requested. If the connection is non-pooled an attempt is made to create the connection. If successful a connection will be created and used. If the connection is pooled and only the single connection was destroyed, another connection is reserved from the pool and used. If the pool was cleared, the pool will attempt to fill to the minimum number of connections specified for the pool. If successful a connection will be reserved and used.

If the transient error still exist at any stage of the pool regeneration or connection creation you should throw an AdapterConnectionException. This will end the current retry operation and Integration Server will start another retry execution or end the retry loop if the maximum attempts have been made.

Retry Mechanisms

The retry facility is provided by Integration Server. The facility depends on what adapter construct detected the failure. A retry is configured with a retry count and a backoff time. Retry count sets the number of times the action will be tried before being considered a fatal error. The backoff time is the time waited between retries. Note that throwing a non-retryable exception on any retry iteration will be treated as a fatal exception and the retries will stop.

The following is a list of adapter objects and how retry behavior applies to each:

- **Connection Pools**. Retry is possible at pool startup. If the backend system is down and the user tries to enable the pool from webMethods Integration Server Administrator, the pool startup will retry until the number of retries is exhausted or the retryable exception goes away. If retries are exhausted, the pool will become disabled. Retries also occur when the pool is created during server startup.
- **Listeners**. Retry is possible during listener startup or during execution. If the number of retries is exhausted, the listener will become disabled. By default, if during execution the Listener Notification produces an AdapterConnectionException, it will cause the corresponding listener to go into retry mode.
- **Adapter Services**. Integration Server provides retry mechanism on services invoked from triggers or flow invokes. An AdapterConnectionException caught from the adapter services

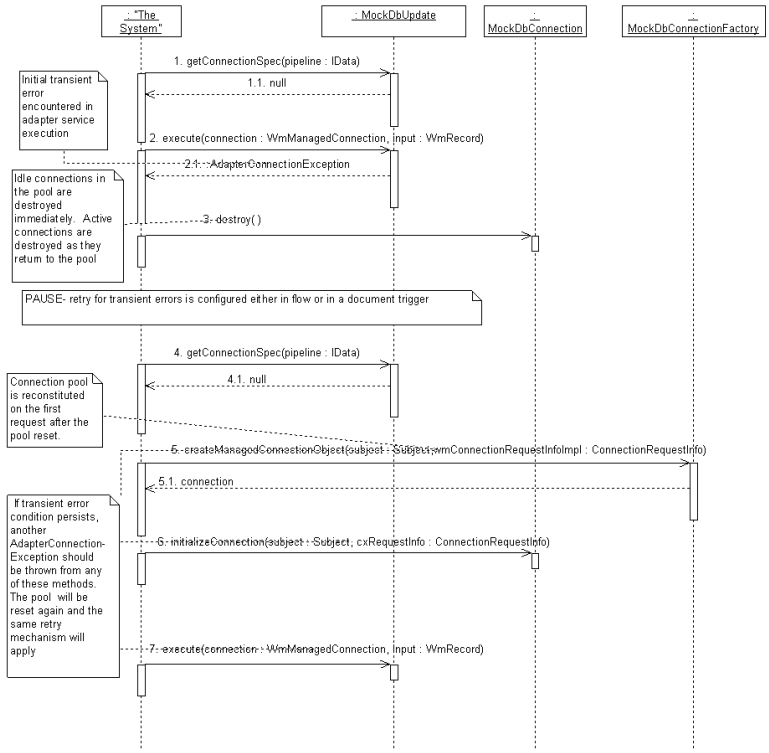
execute() method, will be re-thrown as an ISRuntimeException. This exception is recognized by Integration Server as a transient error and the retry cycle will occur.

- **Polling Notifications**. The polling notification retry works differently then the other components. If a transient error is detected during the schedule execution the proper connection cleansing occurs. The next scheduled interval acts as the retry back off time. However, there is no maximum number of attempts. The notification will remain enabled and will fire on its next scheduled interval.

Putting It All Together…

The following diagram shows the interactions with an adapter during a typical adapter-service retry scenario.

The transient error condition is initially detected in the adapter service's execute method (step 2) whereupon the adapter throws an AdapterConnectionException (step 2.1). This resets the connection pool by destroying all idle connections immediately and any active connections as they return to the connection pool (step 3). Once the connection pool is reset, the error will be sent to the Integration Server service invocation logic as a retryable exception.



If so configured, and after an appropriate timeout period, the adapter service is once again invoked (beginning at step 4). Since the connection pool was emptied after the AdapterConnectionException, a new connection will be created for this service invocation (steps 5 and 6). If a transient error condition still exists, another AdapterConnectionException should be thrown from either the

createManagedConnectionObject() or initializeConnection() call. In that event the pool will once again be reset (step 3) and the retry logic used again.

Run Time Connection Allocation for Adapter Services

When an adapter service is invoked, either directly or from a flow service, the Integration Server adapter run time provides a connection object to the adapter service implementation (the WmAdapterService.execute() method). This section describes how connections are retrieved and managed and how to dynamically control the type of connection used for each service invocation.

At run time, all connection activity for adapter services is performed inside a transaction context that holds references to connections used while the context is open. (This is true regardless of whether the referenced connections are transacted.) There is an implicit transaction context that begins at the invocation of a top-level flow service (such as an HTTP invocation of an Integration Server service) and continues until that top-level service exits. Additional contexts may be created using the pub.art.transaction:startTransaction and ended using pub.art.transaction:commitTransaction or pub.art.transaction:rollbackTransaction. For more information on using these services, see the *webMethods Integration Server Built-In Services Reference* for your release.

When the Integration Server adapter run time retrieves a connection from a connection pool for use by an adapter service, a reference to that connection is placed in the transaction context, and the connection is not returned to the pool until the transaction context is closed. If another adapter service call is made within the transaction context, Integration Server will first determine whether a connection from the required connection pool and partition is in the context; if so, Integration Server will use the connection from the transaction context to the adapter service instead of requesting another from the connection pool.

When a connection is requested from a particular connection pool, the request may identify a partition in the form of an adapter-generated ConnectionRequestInfo object. A connection pool partition is a logical division within a given connection pool where connection objects in different partitions are used at different times in an adapter-defined way. If no ConnectionRequestInfo object is provided in the request, then the default (or null) partition is used. If the pool has an available connection in the specified partition, it marks that connection as busy and returns the connection to the caller. If not, (and the pool is not full,) then the pool requests a new connection from the associated connection factory, including any provided ConnectionRequestInfo object.

Dynamically Selecting a Connection Node

Each connection node should be used to access a single physical resource. In some integration environments, similar functionality is available on multiple physical resources. In these cases, a single adapter service node may be used to access those resources by dynamically specifying which connection node to use for a particular service invocation.

The connection node to be used for a particular invocation is determined as follows:

1. The adapter may specify a connection name by overriding the default implementation of the WmAdapterService.getConnectionSpec() method to return a WmConnectionSpec object containing the connection name. For more information on using WmConnectionSpec objects, see "Extending WmConnectionSpec" on page 171.

2. If the adapter does not specify a connection using getConnectionSpec(), the connection name may be specified on the pipeline in the $connectionName field. Integration Server will only check for a value in $connectionName if the field is part of the service's input signature. For more information on exposing this field in the service signature, see the javadoc for WmDescriptor.showConnectionName().

3. If a connection node name still has not been specified, the service's default connection is used. The default connection is generally the connection that was used when the adapter service was created, unless it has been changed using either the pub.art.service:setAdapterServiceNodeConnection or the wm.art.dev.service:updateAdapterServiceNode service.

   To update a service node, in Designer you must either lock the node for editing or check out the node. When the Integration Server watt.server.ns.lockingMode property is set to `system`, you must obtain Write ACL access to the service node that you want to edit before updating the node. For information about obtaining Write ACL access, see the *webMethods Service Development Help* for your release.

Partitioned Connection Pools

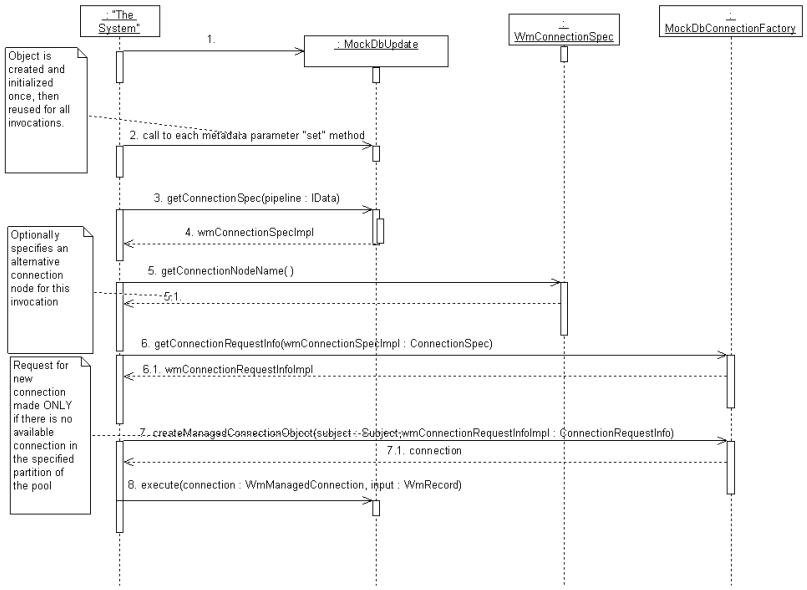
While a connection node defines a general set of connections to a given back-end resource, it is sometime necessary to connect to the backend using attributes that are specific to a particular operation or data set. For example, the backend may require that a connection be established with a specific set of user credentials in order to update a given record. If the number of unique attributes is small, it may be possible to define a connection pool with each set of attributes and select the appropriate pool at run time based on the service being called and/or the data being manipulated. When this is not practical, an adapter should implement connection pool partitioning.

A connection pool partition is a logical division within a given connection pool where connection objects in different partitions are used at different times in an adapter-defined way. Typically, connections in different partitions have different permissions, and are often associated with different users. However, the use of partitions and how they are segregated is entirely determined by the adapter implementation. Connection objects are assigned to a particular partition at the time they are created and remain in the same partition for the life of the object.

When a connection pool is started, it is initialized with connections in the default (null) partition. Additional partitions are created and populated as connections from those partitions are requested.

The process of requesting a connection from a particular partition begins in the WmAdapterService.getConnectionSpec() method. Adapter writers should extend the WmConnectionSpec class to capture any context information necessary to determine the required partition. This may include information from the invocation pipeline, as well as information about the service being invoked, including metadata parameter settings.

The WmConnectionSpec object returned by the service implementation is then passed to the getConnectionRequestInfo() method of the connection factory associated with the selected connection node. (See "Enabling Connection Nodes" on page 63 for information on selecting a connection node for an invocation. Note that the connection node need not be identified in the WmConnectionSpec object.) The connection factory is then required to identify the connection pool partition based on the information in the WmConnectionSpec object.



Implementing Partitioned Connection Pools

In order to support partitioned connection pools in an adapter, the adapter developer must do the following:

- Create a class that defines the partition and extends and implements com.wm.adk.connection.WmConnectionRequestInfo.
- Create a class to hold any required information gathered from an adapter service invocation. This class must extend com.wm.adk.connection.WmConnectionSpec.
- Implement the getConnectionRequestInfo() method in the adapter's WmManagedConnectionFactory implementation
- Implement the getConnectionSpec() method in the adapter's WmAdapterService implementation.

To demonstrate an implementation, we will use the fictional MegaBank corporation. MegaBank is an aggressive company, regularly acquiring and absorbing the customer base of smaller banks. A customer acquired in this way is assigned a new MegaBank user name with which he can access his accounts as well as other services provided by the larger bank. Since MegaBank has adapters for most major banking systems, it is able to seamlessly integrate the new bank's system very quickly, by mapping the customer's new MegaBank user name with login information already present in the acquired bank's system. Using partitioned pools, connections are established using authentication information already known to the backend.

Extending WmConnectionRequestInfo

A ConnectionRequestInfo object defines a partition both within the connection pool and in WmManagedConnectionFactory.createManagedConnectionObject() when a new connection is

created. The connection pool organized its connections based on the ConnectionRequestInfo object used when the connection was created. In this case, ConnectionRequestInfo objects X and Y are considered the same if (X.hashCode() == Y.hashCode() && X.equals(Y)) The remainder of the ConnectionRequestInfo implementation is defined by the adapter.

In the MegaBank example, a partition is defined by the userId with which the back-end connection is established. In order to ensure that the connection pool properly recognizes objects that refer to the same partition, the example below implements the hashCode() and equals() methods so that they only refer to the userId String.

The final abstract method that must be implemented is getLogableName(). This method is used to provide a partition name that can be recorded in Integration Server logs. In this example, it is also used to correlate the MegaBank user name with the ID used to access the backend. Care should be taken in implementing this method to avoid exposing any sensitive data (such as passwords).

```
public class MbConnectionRequestInfo extends WmConnectionRequestInfo
{

     private final String userId;
     private final String mbUserName;
     private final Credentials credentials;
     private final String logName;

     /**
      * Sole constructor. verifies all fields populated.
      *
      * @param mbUserName - common user name used across the integration
      * @param userId - name of user as known to the backend system
      * @param credentials - credentials required for this user to access
      *                      the backend system
      * @throws IllegalArgumentException - if any argument is null
      */
     public MbConnectionRequestInfo(String mbUserName, String userId,
             Credentials credentials) throws IllegalArgumentException
     {
         if (mbUserName == null || userId == null || credentials == null)
         {
             throw new IllegalArgumentException();
         }
         this.userId = userId;
         this.mbUserName = mbUserName;
         this.credentials = credentials;
         this.logName = mbUserName + "(" + userId + ")";
     }

     public Credentials getCredentials()
     {
         return credentials;
     }
     public String getMbUserName()
     {
         return mbUserName;
     }
     public String getUserId()
     {
         return userId;
     }
```

```
    /**
     * Name used by the connection pool when creating log entries relevant
     * to the partition identified by this object.
     */
    public String getLoggableName()
    {
        return logName;
    }
    /**
     * In a ConnectionRequestInfo object the hashCode of objects that
     * identifythe same partion must be the same, so we use only the
     * hashcode of the userId String.
     */
    public int hashCode()
    {
        return this.userId.hashCode();
    }
    /**
     * If two ConnectionRequestInfo objects identify the same partition,
     * then the equals method must return true. We compare the userIds of
     * the two object rather than allow the default equals implementation
     * which only checks if they are the same object instance.
     * @param obj - object being compared to this.
     */
    public boolean equals(Object obj)
    {
        boolean isEqual = false;
        if(obj instanceof MbConnectionRequestInfo)
    {
        isEqual = this.userId.equals(((MbConnectionRequestInfo)obj).userId);
    }
    return isEqual;
}
```

Extending WmConnectionSpec

A WmConnectionSpec object is used during an adapter service invocation to identify a connection node and/or to hold any contextual information from the service configuration and invocation pipeline that is needed to identify the partition of the connection. Accessors for the connection node name are provided in the base class. Contextual information need for partition definition is specific to the adapter and must be implemented in an adapter-defined subclass.

For the MegaBank example, each customer is assigned a username that allows access to all of the bank's services. This value must be cross-referenced to obtain the name by which that user is known on the specific back-end system. This cross-referencing will be delegated to the WmConnectionFactory.getConnectionRequestInfo() implementation described later in this document. For now, we simply need a container to hold the information.

```
public class MbConnectionSpec extends WmConnectionSpec
{
 private String mbUserName;
 public MbConnectionSpec()
 {
  super();
 }
 public String getMbUserName()
 {
```

```
  return mbUserName;
 }
 public void setMbUserName(String mbUserName)
 {
  this.mbUserName = mbUserName;
 }
}
```

Updating the Connection Factory

To complete support for the partitioned connection pool feature, the adapter's WmManagedConnectionFactory implementation must be able to produce valid WmConnectionRequestInfo objects and it must be able to use those objects to create connections with the required characteristics.

As shown in the sample MegaBank code below, the implementation of getConnectionRequestInfo() uses information from the provided ConnectionSpec object to produce a valid ConnectionRequestInfo object. In this case, the MegaBank user name is used as a key to lookup a set of user ID and credentials appropriate for the backend being accessed by this connection. (Details of how this lookup is implemented are adapter-specific and outside the scope of this document.)

```
/**
* Produce a ConnectionRequestInfo object based on the provided ConnectionSpec
* object. If an mbUserName is provided, lookup the name to use on the backend
* for this connection pool.
*/
public ConnectionRequestInfo getConnectionRequestInfo(ConnectionSpec spec)
{
        MbConnectionRequestInfo partitionDef = null;
        if(spec != null && spec instanceof MbConnectionSpec)
        {
            synchronized(this)
            {
                try
                {
                    String mbUserName = ((MbConnectionSpec)spec).getMbUserName();
                  lookupUser(mbUserName);
                  partitionDef = new MbConnectionRequestInfo(mbUserName,
                            this.backendUserID, this.credentials);
                }
                catch(IllegalArgumentException ex)
                {
                   // no partition info available for this user. Swallow the
                   // exception and return null
                }
          }
     }
         return partitionDef
}
```

When the connection pool needs a new connection belonging to a particular partition, the ConnectionRequestInfo object returned above is passed to the createManagedConnectionObject method. In the sample code shown below, this information is used to override default logon information that is otherwise established based on metadata parameter settings. Note that all

createManagedConnectionObject implementations must be able to establish default-partition connections when null is passed in the cxRequestInfo argument.

```
/**
* Create a connection using userId and credentials from the provided
* connectionRequestInfo object, if provided. If connectionRequestInfo is
* not provided use the default userId and credentials information established
* at node startup from metadata parameters.
*/
public WmManagedConnection createManagedConnectionObject(Subject subject,
        ConnectionRequestInfo cxRequestInfo) throws ResourceException
{
     String connUser;
     Credentials connCredentials;
     if (cxRequestInfo != null && cxRequestInfo instanceof
       MbConnectionRequestInfo)
     {
         connUser = ((MbConnectionRequestInfo)cxRequestInfo).getUserId();
         connCredentials =
         ((MbConnectionRequestInfo)cxRequestInfo).getCredentials();
     }
     else
     {
         connUser = this.defaultUserId;
         connCredentials = this.defaultCredentials;
     }
     return new MbConnection(connUser, connCredentials);
}
```

A Alternative Approaches to Metadata

- Overview 176
- Implementing Metadata Parameters Using External Classes 176
- An Alternative Approach to Organizing Resource Domains 176
- Using Resource Bundles with Resource Domain Values 180

Overview

This chapter describes other capabilities supported by the ADK that are useful, but not required, for implementing an adapter.

Implementing Metadata Parameters Using External Classes

A basic model, by which all adapter metadata parameters are specified, is described in "webMethods Metadata Parameters" on page 58. The server derives a parameter's name and data type from the name of the accessor methods in the class to which the parameter applies.

If a class contains a conforming accessor method that uses an object data type other than one of the listed data types, the adapter interprets that object as being an external container for the metadata parameters. For example, if a WmAdapterService class contains an accessor method such as:

```
SetParameters(MyServiceParameters value);
```

then the server considers MyServiceParameters to be an external class that contains accessor methods. The server introspects the MyServiceParameters class, and derives metadata parameter names from it. MyServiceParameters must support a default (no argument) constructor.

In addition, this feature changes the name of the parameter string used in the descriptor methods (as well as the resource bundle) by prefixing it with the name derived from the method that implements the indirection. Continuing with the previous example, if MyServiceParemeters includes a method such as setFoo(String value), then the string that referenced this parameter would be:

```
parameters.foo
```

where parameters is derived from setParameters, and foo is derived from setFoo.

An Alternative Approach to Organizing Resource Domains

The model described in this section provides an alternative way of organizing resource domain information, such that the resource domain implementation is contained within each adapter service or notification class that uses the resource domain, rather than within your WmManagedConnection implementation (as described in "Resource Domains" on page 73).

To implement this approach, you must update a string array (usually in the connection factory) containing the class name for each service or notification type added to an adapter. No additional changes are required for the connection factory or connection implementation. These service and notification classes must implement the ResourceDomainHandler interface, which enables the classes to manage their own resource domain functionality. A complete listing of the ResourceDomainHandler interface is provided below.

Note:
The ResourceDomainHandler interface is not delivered as part of the ADK.

Note:

An example of a notification that implements this interface is provided in "Defining a WmAsyncListenerNotification Implementation Class" on page 143.

```
package com.mycompany.adapter...;
import com.wm.adk.connection.WmManagedConnection;
import com.wm.adk.metadata.*;
import com.wm.adk.error.*;
public interface ResourceDomainHandler
{
     /**
     * Implements resource domain lookups using the provided connection. Refer to
     * the method of the same name in com.wm.adk.connection.WmManagedConnection.
     *
     * @param connection
     * @param resourceDomainName
     * @param values
     * @return ResourceDomainValues[]
     * @throws AdapterException
     */
    public  ResourceDomainValues[] adapterResourceDomainLookup(
               WmManagedConnection connection,
               String resourceDomainName,
               String[][] values) throws AdapterException;
    /**
     * Implements Adapter check values using the provided connection. Refer to
      * the method of the same name in com.wm.adk.connection.WmManagedConnection.
     *
     * @param connection
     * @param resourceDomainName
     * @param values
     * @param testValue
     * @return Boolean
     * @throws AdapterException
     */
    public Boolean adapterCheckValue( WmManagedConnection connection,
                                      String resourceDomainName,
                                      String[][] values,
                                          String testValue) throws AdapterException;

    /**
     * Implements resource domain registrations specific to a particular service
     * or notification. Refer to the method of the same name in
     * com.wm.adk.connection.WmManagedConnectionFactory.
     *
     * @param connection
     * @param access
     * @throws AdapterException
     */
    public void registerResourceDomain(WmManagedConnection connection,
                                       WmAdapterAccess access)throws
                                       AdapterException;
}
```

To use this model, you must have a list of services and notifications that implement the interface. Since the connection factory must already register its services, the connection factory is a logical place to create these lists. The lists are then passed to the connections that the connection factory creates. For example:

```
public class SimpleConnectionFactory extends WmManagedConnectionFactory
{
    private static final String[] _supportedServiceTemplates = {
                                        SimpleService.class.getName(),
                                        TestService.class.getName(),
                                        CopySLN.class.getName(),
                                        Copy2SLN.class.getName()};
    private static final String[] _supportedNotificationTemplates = {
                                        SimpleNotification.class.getName(),
SessionLogListenerNotification.class.getName()};
  // add metadata parameter support here
   public WmManagedConnection createManagedConnectionObject(
                  javax.security.auth.Subject subject,
                  javax.resource.spi.ConnectionRequestInfo cxRequestInfo)
   {
      ArrayList templateList = new ArrayList(
      Arrays.asList(_supportedServiceTemplates));
      templateList.addAll(Arrays.asList(_supportedNotificationTemplates));
      String [] listArg = new String[templateList.size()];
      templateList.toArray(listArg);
      return new SimpleConnection(...,listArg);
   }
   public void fillWmDescriptor(WmDescriptor d,Locale l) throws
                               AdapterException
   {
   ...
   }
   public void fillResourceAdapterMetadataInfo(
              ResourceAdapterMetadataInfo info,
              Locale locale)
   {
        String[] templateList = _supportedServiceTemplates;
        for (int i = 0; i < templateList.length;i++)
        {
            info.addServiceTemplate(templateList[i]);
        }
   }
}
```

The connection implementation then uses the service name it receives in the adapterResourceDomainLookup, and adapterCheckValue calls to forward those requests to the appropriate service or notification class:

```
public class SimpleConnection extends WmManagedConnection
{
    private String[] _resourceHandlerList;
    public SimpleConnection(...,String[] resourceHandlerList)
    {
        _resourceHandlerList = resourceHandlerList;
...
    }
    public void registerResourceDomain(WmAdapterAccess access)throws
                                                    AdapterException
    {
        try
        {
            Class serviceClass;
            ResourceDomainHandler serviceObject;
            for (int i = 0;i < _resourceHandlerList.length;i++ )
```

```
            {
                serviceClass = Class.forName(_resourceHandlerList[i]);
                serviceObject =
                         (ResourceDomainHandler)serviceClass.newInstance();
                serviceObject.registerResourceDomain(this,access);
            }
        }
        catch (Throwable t)
        {
            throw MyAdapter.getInstance().createAdapterException(9999,t);
        }
    }
    public Boolean adapterCheckValue( String serviceName,
                                      String resourceDomainName,
                                      String[][] values,
                                      String testValue)throws
                                                    AdapterException
    {
        Class serviceClass;
        ResourceDomainHandler serviceObject;
        try
        {
            serviceClass = Class.forName(serviceName);
            serviceObject =
                          (ResourceDomainHandler)serviceClass.newInstance();
        }
        catch (Throwable t)
        {
            throw
                MyAdapter.getInstance().createAdapterException(9999,t);
        }
        return
serviceObject.adapterCheckValue(this,resourceDomainName,values,testValue);
    }
    public  ResourceDomainValues[] adapterResourceDomainLookup(
                                    String serviceName,
                                    String resourceDomainName,
                                    String[][] values) throws AdapterException
    {
        Class serviceClass;
        ResourceDomainHandler serviceObject;
        try
        {
            serviceClass = Class.forName(serviceName);
            serviceObject =
                          (ResourceDomainHandler)serviceClass.newInstance();
        }
        catch (Throwable t)
        {
            throw
                MyAdapter.getInstance().createAdapterException(9999,t);
        }
        return
serviceObject.adapterResourceDomainLookup(this,resourceDomainName,values);
    }
...
}
```

Using Resource Bundles with Resource Domain Values

A resource bundle typically contains all display strings and messages used by the adapter at runtime and at design time. A resource bundle is also specific to a particular locale. Resource bundles enable you to internationalize an adapter quickly, without having to change any code in the adapter. For more information, see "Creating Resource Bundles" on page 36

Adapters may also make explicit use of a resource bundle to localize other data, such as resource domain values, especially if those values are known at development time (for example, a list of known record status values). (For more information about resource domain values, see "Populating Resource Domains with Values" on page 77.) In those cases, the strategy for key composition is left to your discretion. However, be careful when a value that was localized needs to be understood at other points in the code (for example, if (this.getStatus() != "active")).

For example, suppose an adapter service includes a metadata parameter called status, which can be set to a value of active or inactive, and that these values should be localized so that active and inactive appear in the language of the current user. If the run-time locale might be different from the locale of the design-time client, then the adapter needs to know the language that was used when the value of status was set. So essentially, instead of code that says 'if (this.getStatus() != "active")', the code must say 'if (this.getStatus() != inLanguageOfDesignTimeClient("active"))'. The problem becomes more complicated if the adapter service node is managed (edited) in more than one locale. If a resource domain lookup depends on a value that is localized, and that value was set by a client in a different locale than the current client, then you need to say 'if (this.getStatus() != inLanguageUsedWhenValueWasSet("active"))'.

To support this functionality, the ADK inserts a special metadata property, called designTimeLocale, in each adapter service node and notification node. The value of designTimeLocale is set when the node is created. Adapter users update the value (in the node) when they press the "Reload values from the adapter" button on the Designer toolbar (which appears while users view an adapter service or notification node). An adapter that uses this property can then present resource domain values in the appropriate client locale when the node is created, and understand the value at run time regardless of what locale the server may use. If a Japanese client wants to edit a node that was configured by an English client, the initial presentation in the Adapter Service Editor or Adapter Notification Editor will reflect the English values currently stored in the node. When a user presses the "Reload values from the adapter" button, the server changes the designTimeLocale to Japanese, and performs all resource domain lookups using the new locale. From the adapter user's perspective, all localized values change from English to Japanese. (This example assumes that the adapter includes both an English and a Japanese resource bundle.)

To support localized resource domain values, include the property name ADKGLOBAL.DESIGN_LOCALE_PROPERTY in the resource domain dependency list of any parameter that uses a resource domain with localized resource domain values. Include this property name for any parameters where the lookup depends on a parameter containing localized values, as well. For example:

```
d.setResourceDomain( "status", "statusLookup", new
                     String[]{ADKGLOBAL.DESIGN_LOCALE_PROPERTY} );
d.setResourceDomain( "statusDescription","statusDescriptionLookup", new
                     String[]
                     { "status",ADKGLOBAL.DESIGN_LOCALE_PROPERTY } );
```

You must present designTimeLocale as a string (derived from Locale.toString) so the value is made available as a resource domain value. To convert this string to a Locale object, use AdapterUtil.parseLocaleString.

Within the implementation of the statusLookup, the adapter uses the designTimeLocale value provided to access the create a list of localized string values:

```
if (resourceDomainName.equals("statusLookup"))
{
AdapterResourceBundleManager ar =
         MyAdapter.getInstance().getAdapterResourceBundleManager();
Locale lookupLocale = AdapterUtil.parseLocaleString(values[0][0]);
for (int i =0; i< statusNames.length;i++)
{
    try {
       displayNames[i] =
                      ar.getStringResource(statusNames[i], lookupLocale);
    }
    catch(Throwable t)
    {
       displayNames[i] = statusNames[i];
    }
  }
  ResourceDomainValues rdv = new
                 ResourceDomainValues(resourceDomainName,displayNames);
  rdv.setComplete(true); // allow user edit
  return new ResourceDomainValues[] {rdv};
}
```

At run time, the adapter uses the getDesignTimeLocale method to retrieve the locale, and to interpret the value of localized parameters. For example:

```
Locale lookupLocale =
            AdapterUtil.parseLocaleString(this.getDesignTimeLocale());
AdapterResourceBundleManager ar =
            MyAdapter.getInstance().getAdapterResourceBundleManager();
if (this.getStatus.equals(ar.getStringResource("active",lookupLocale)))
{
...
```

For more information, see the Javadoc for the following:

- com.wm.adk.ADKGLOBALS.DESIGN_TIME_LOCALE_PROPERTY
- com.wm.adk.cci.interaction.WmAdapterService.getDesignTimeLocale()
- com.wm.adk.notification.WmNotification.getDesignTimeLocale()
- com.wm.adk.util.AdapterUtil.parseLocaleString()

B Integration Server Transaction Support

- Overview 184
- Simple Transactions 184
- More Complex Transactions 185
- Implicit Transaction Usage Cases 186
- Explicit Transaction Usage Cases 187
- Built-In Services For Explicit Transactions 191
- Transaction Error Situations 194
- Specifying Transaction Support in Connections 195

Overview

This section describes how webMethods Integration Server supports transactions. Integration Server considers a transaction to be one or more interactions with one or more resources that are treated as a single logical unit of work. The interactions within a transaction are either all committed or all rolled back. For example, if a transaction includes multiple database inserts, and one or more inserts fail, all inserts are rolled back.

Integration Server supports the following kinds of transactions:

- A *local transaction*, which is a transaction to a resource's local transaction mechanism
- An *XAResource transaction*, which is a transaction to a resource's XAResource transaction mechanism

Integration Server can automatically manage both kinds of transactions, without requiring the adapter user to do anything. Integration Server uses a container-managed (implicit) transaction management approach based on the JCA standard, and also performs some additional connection management. This is because adapter services use connections to create transactions. For more information, see "Implicit Transaction Usage Cases" on page 186.

However, there are cases where adapter users need to explicitly control the transactional units of work. Examples of these cases are provided in "Explicit Transaction Usage Cases" on page 187.

To support transactions, Integration Server relies on a built-in transaction manager. The transaction manager is responsible for beginning and ending transactions, maintaining a transaction context, enlisting newly connected resources into existing transactions, and ensuring that local and XAResource transactions are not combined in illegal ways.

Beginning with Integration Server 8.0, the Transaction Manager also manages operations performed by a transacted JMS trigger, or a built-in JMS service that uses a transacted JMS connection alias.

Important:
You cannot create steps and trace a flow that contains a transacted adapter service.

Note:
If you interact with a resource that does not support transactions, Integration Server will not create a transaction for it.

To specify which kind(s) of transactions to support in your adapter, see "Specifying Transaction Support in Connections" on page 195.

Simple Transactions

The simplest Integration Server transaction scenario is a flow service (or a Java service) that invokes one adapter service that interacts with one resource. For example, the transaction might perform a database insert.

Integration Server executes this transaction without requiring adapter users to perform any transaction management, as follows.

1. Integration Server invokes the request as follows:

   a. The adapter service obtains a connection from the connection pool, creates a transaction, and enlists the connection in the current transaction.

   b. The adapter service performs the database insert. If the insert fails, a ServiceException is thrown.

2. Integration Server informs the transaction manager that the service request has completed as follows:

   - If the service request succeeds, the transaction manager commits the current transaction. If the commit fails, the transaction manager throws an exception that causes the service request to fail, and a ServiceException is returned to the adapter user.
   - If the service request fails, the transaction manager rolls back the current transaction.

Note that the commit or rollback occurs after the service has completed, but before any response is sent to the client that invoked the service. This is because if the transaction commit fails, the service itself must fail. So, the transaction notification essentially becomes the last step of the service (as opposed to occurring after the service has already completed).

More Complex Transactions

Now suppose that a flow service (or a Java service) invokes multiple adapter services. These adapter services might interact with a single resource (such as two services that perform two inserts into a single database) or with multiple resources (such as services that synchronize a database and an ERP system).

You can have one or more connections to a single resource. If multiple connections are used, the server enlists each connection in the transaction.

When the server request is complete, Integration Server notifies the transaction manager, which closes all enlisted connections and transactions. If the service request is successful, the server commits all transactions automatically (using a two-phase commit if multiple resources are used). If the service request returns an error, the server rolls back the transaction and returns an error to the Integration Server; this causes the service to return an error.

As mentioned previously, there are cases where adapter users need to explicitly control the transactional units of work. Examples of these cases are provided in "Explicit Transaction Usage Cases" on page 187.

Note:
If a transaction accesses multiple resources, and more than one of the resources only supports local transactions, the integrity of the transaction cannot be guaranteed. For example, if the first resource successfully commits, and the second resource fails to commit, the first resource interaction cannot be rolled back; it has already been committed. To help prevent this problem, Integration Server detects this case when connecting to more than one resource that does not support two-phase commits. It throws a run-time exception and the service execution fails.

Implicit Transaction Usage Cases

Implicit transactions are handled automatically by Integration Server. For a flow to be managed implicitly, it can contain one of the following:

- One local transaction, interacting with one resource
- One or more XAResource transactions; each transaction can interact with one or more resources
- One or more XAResource transactions and one local transaction

If a flow contains multiple local transactions, the adapter user must explicitly control the transactional units of work, as described in "Explicit Transaction Usage Cases" on page 187.

Following are examples of implicit transactions.

One Local Transaction

In this example, a flow with two adapter services interacts with the same local transaction resource. The flow performs two inserts into two tables of a database:

```
BEGIN FLOW
INVOKE insertDatabase1TableA // Local Transaction Resource1
INVOKE insertDatabase1TableB // Local Transaction Resource1
END FLOW
```

Integration Server starts the transaction when insertDatabase1TableA is invoked. It opens a connection to the resource, enlists it in the transaction, and performs the insert into TableA. When insertDatabase1TableB is invoked, Integration Server reuses the same connection to insert data into TableB. When the request is complete, Integration Server closes the connection and commits the transaction.

The following flow is *illegal* because it tries to interact with *two* local transaction resources as follows:

```
BEGIN FLOW
INVOKE insertDatabase1TableA // Service for Resource1
INVOKE insertDatabase2TableA // Service for Resource2
END FLOW
```

Three XAResource Transactions

The following flow is valid because a flow can contain any number of XAResource transactions.

```
BEGIN FLOW
INVOKE insertDatabase1TableA // XAResource Transaction Resource1
INVOKE insertDatabase2TableA // XAResource Transaction Resource2
INVOKE insertDatabase3TableA // XAResource Transaction Resource3
END FLOW
```

One Local Transaction and One XAResource Transaction

Continuing with the previous case, notice this additional insert to a different database that accepts XAResource transactions as follows:

```
BEGIN FLOW
INVOKE insertDatabase1TableA // Local Transaction Resource1
INVOKE insertDatabase1TableB // Local Transaction Resource1
INVOKE insertDatabase2TableA // XAResource Transaction Resource1
END FLOW
```

When Integration Server invokes insertDatabase2TableA, a transaction is already in progress with the first database enlisted. It then establishes a second connection (to Database2), enlists the new connection in the XAResource transaction, and performs the insert to tableA.

When the request is complete, Integration Server closes both connections and the Transaction Manager performs a local commit for the non-XAResource and then a two-phase commit for the XAResource enlisted in the transaction.

Explicit Transaction Usage Cases

To include multiple local transactions in a single flow, adapter users must explicitly start and end each transaction except the first one.

Depending on what the flow needs to accomplish, adapter users may explicitly start and end XAResource transactions as well. This way, they can create a flow that includes multiple local transactions and multiple XAResource transactions.

To support this, Integration Server provides the following built-in services:

- pub.art.transaction.startTransaction (see "WmART.pub.art.transaction:startTransaction" on page 193
- pub.art.transaction.commitTransaction (see "WmART.pub.art.transaction:commitTransaction" on page 192
- pub.art.transaction.rollbackTransaction (see "WmART.pub.art.transaction:rollbackTransaction" on page 192
- pub.art.transaction.setTransactionTimeout (see "WmART.pub.art.transaction:setTransactionTimeout" on page 193

For example, the following flow includes a local transaction nested within another local transaction:

```
BEGIN FLOW                      // start transaction 1
.
.
.
    INVOKE startTransaction(2)      // start transaction 2
    .
    .
    .
    INVOKE commitTransaction(2)     // commit transaction 2
END FLOW                        // commit transaction 1
```

A nested transaction *must* adhere to the same rules that apply to container-manager transactions. That is, a nested transaction can contain *one* of the following:

- One local transaction, interacting with one resource
- One or more XAResource transactions; each transaction can interact with one or more resources
- One or more XAResource transactions and one local transaction

Following are some examples of explicit transactions.

Two Local Transactions

To make this flow work properly, explicitly start and commit the nested local transaction, using the startTransaction and commitTransaction services as follows:

```
BEGIN FLOW                    // start transaction 1
INVOKE interactWithResourceA  // service for transaction 1

   INVOKE startTransaction(2)     // start transaction 2
   INVOKE interactWithResourceB   // service for transaction 2
   INVOKE commitTransaction(2)    // commit transaction 2

END FLOW                      // commit transaction 1
```

The flow executes as follows:

1. When interactWithResourceA is invoked, Integration Server starts transaction 1 and enlists ResourceA.
2. Transaction 2 executes as follows:
   a. When startTransaction(2) is invoked, Integration Server starts a new, nested transaction.
   b. When interactWithResourceB is invoked, ResourceB is enlisted in transaction 2.
   c. When commitTransaction(2) is invoked, the connection to ResourceB is closed, and transaction 2 is committed. At this point, only the work done on ResourceB is committed; transaction 1 is still open, and the work done with ResourceA is not yet committed.
3. When the flow ends, Integration Server closes the connection for transaction 1 and commits its work to ResourceA.

Note:
Each transaction is a separate unit of work. Transaction 1 could be rolled back (or the commit could fail), while transaction 2 remains committed (or vice versa).

Alternatively, to achieve the same result, you can explicitly start transaction 1 before the adapter service is invoked, and explicitly commit it as follows:

```
BEGIN FLOW
INVOKE startTransaction(1)     // start transaction 1
INVOKE interactWithResourceA   // service for transaction 1
INVOKE startTransaction(2)     // start transaction 2
INVOKE interactWithResourceB   // service for transaction 2
```

```
INVOKE commitTransaction(2)    // commit transaction 2
INVOKE commitTransaction(1)    // commit transaction 1
END FLOW
```

Two XAResource Transactions

The following flow includes two XAResource transactions: one that interacts with ResourceA, and a nested transaction that interacts with ResourceB and ResourceC.

```
BEGIN FLOW                      // start transaction 1
INVOKE interactWithResourceA    // service for transaction 1

   INVOKE startTransaction(2)      // start transaction 2
   INVOKE interactWithResourceB    // service for transaction 2
   INVOKE interactWithResourceC    // service for transaction 2
   INVOKE commitTransaction(2)     // commit transaction 2

END FLOW                        // commit transaction 1
```

The flow executes as follows:

1. When interactWithResourceA is invoked, Integration Server starts transaction 1 and enlists ResourceA.
2. Transaction 2 executes as follows:
   a. When startTransaction(2) is invoked, Integration Server starts a new, nested transaction.
   b. When interactWithResourceB and interactWithResourceC are invoked, both resources are enlisted in transaction 2.
   c. When commitTransaction(2) is invoked, the connections to ResourceB and ResourceC are closed, and transaction 2 is committed. At this point, only the work done on ResourceB and ResourceC is committed; transaction 1 is still open, and the work done with resourceA is not yet committed.
3. When the flow ends, Integration Server closes the connection for transaction 1 and commits its work to ResourceA.

One XAResource Transaction and Two Nested Local Transactions

The following flow includes three transactions: one XAResource transaction that interacts with two resources, and two nested local transactions that interact with one resource each.

```
BEGIN FLOW                       // start XAResource transaction 1
INVOKE interactWithXAResourceA   // service for XAResource transaction 1
INVOKE interactWithXAResourceB   // service for XAResource transaction 2
   INVOKE startTransaction(2)         // start local transaction 1
   INVOKE interactWithLocalResourceA  // service for local transaction 1
   INVOKE commitTransaction(2)        // commit local transaction 1
   INVOKE startTransaction(3)         // start local transaction 2
   INVOKE interactWithLocalResourceB  // service for local transaction 2
   INVOKE commitTransaction(3)        // commit local transaction 2
END FLOW                         // commit XAResource transaction 1
```

The flow executes as follows:

1. When interactWithXAResourceA is invoked, Integration Server starts transaction 1 and enlists XAResourceA.
2. When interactWithXAResourceB is invoked, Integration Server enlists XAResourceB in transaction 1.
3. Transaction 2 is executed as follows:
   a. When startTransaction(2) is invoked, Integration Server starts a new, nested transaction.
   b. When interactWithLocalResourceA is invoked, LocalResourceA is enlisted in transaction 2.
   c. When commitTransaction(2) is invoked, the connection to LocalResourceA is closed, and transaction 2 is committed. At this point, only the work done on LocalResourceA is committed; transaction 1 is still open, and the work done with XAResourceA and XAResourceB is not yet committed.
4. Transaction 3 is executed as follows:
   a. When startTransaction(3) is invoked, Integration Server starts a new, nested transaction.
   b. When interactWithLocalResourceB is invoked, LocalResourceB is enlisted in transaction 3.
   c. When commitTransaction(3) is invoked, the connection to LocalResourceB is closed, and transaction 3 is committed. At this point, only the work done on LocalResourceA and LocalResourceB is committed; transaction 1 is still open, and the work done with XAResourceA and XAResourceB is not yet committed.
5. When the flow ends, Integration Server closes the connection for transaction 1 and commits its work to XAResourceA and XAResourceB.

One XAResource Transaction and One Nested Local and XAResource Transaction

The following flow includes two transactions: one XAResource transaction that interacts with two resources, and one nested transaction that interacts with one local resource and one XAResource.

```
BEGIN FLOW                        // start XAResource transaction 1
INVOKE interactWithXAResourceA    // service for XAResource transaction 1
INVOKE interactWithXAResourceB    // service for XAResource transaction 2

   INVOKE startTransaction(2)           // start transaction 2
   INVOKE interactWithLocalResourceA    // service for transaction 2
   INVOKE interactWithXAResourceC       // service for transaction 2
   INVOKE interactWithLocalResourceA    // service for transaction 2
   INVOKE commitTransaction(2)          // commit transaction 2

END FLOW                          // commit XAResource transaction 1
```

The flow executes as follows:

1. When interactWithResourceA is invoked, Integration Server starts an XAResource transaction 1 and enlists ResourceA.
2. When interactWithResourceB is invoked, Integration Server enlists ResourceB in transaction 1.
3. Transaction 2 is executed as follows:
    a. When startTransaction(2) is invoked, Integration Server starts a new, nested transaction.
    b. When interactWithLocalResourceA is invoked, LocalResourceA is enlisted in transaction 2.
    c. When interactWithXAResourceC is invoked, XAResourceC is enlisted in transaction 2.
    d. When interactWithLocalResourceA is invoked, LocalResourceA is enlisted in transaction 2.
    e. When commitTransaction(2) is invoked, the connection to both resources of transaction 2 is closed, and transaction 2 is committed. At this point, only the work done on LocalResourceA and XAResourceC is committed; transaction 1 is still open, and the work done with XAResourceA and XAResourceB is not yet committed.
4. When the flow ends, Integration Server closes the connection for transaction 1 and commits its work to XAResourceA and XAResourceB.

Built-In Services For Explicit Transactions

Use the built-in services described in this section to manage explicit transactions for your Adapter Development Kit services.

Explicit transactions are transactions that you manually control within flow services through the use of built-in services. Implicit transactions are automatically handled by the Integration Server's transaction manager. When you define an explicit transaction, it is nested within the implicit transactions that are controlled by the transaction manager. You can have more than one explicit transaction defined within an implicit transaction. You can also nest explicit transactions within each other.

Any flow service steps found between a WmART.pub.art.transaction:startTransaction service and either a WmART.pub.art.transaction:commitTransaction service or a WmART.pub.art.transaction:rollbackTransaction service are part of an explicit transaction rather than the implicit transaction.

Within both implicit and explicit transactions, you cannot have multiple connections with a transaction type of LOCAL_TRANSACTION because you will not be able to rollback the first LOCAL_TRANSACTION adapter service after it is committed. Use the built-in services to define explicit transactions to prevent you from inadvertently committing transactions if you need to rollback the transaction.

WmART.pub.art.transaction:commitTransaction

This service commits an explicit transaction. It must be used in conjunction with the WmART.pub.art.transaction:startTransaction service. If it does not have a corresponding WmART.pub.art.transaction:startTransaction service, your flow service will receive a run-time error.

For more information about implicit and explicit transactions, see "Built-In Services For Explicit Transactions" on page 191.

Input Parameters

commitTransactionInput	**Document.**A document that contains the variable *transactionName*, described below.
transactionName	**String.**Used to associate a name with an explicit transaction. The *transactionName* must correspond to the *transactionName* in any WmART.pub.art.transaction:startTransaction or WmART.pub.art.transaction:rollbackTransaction services associated with the explicit transaction.

WmART.pub.art.transaction:rollbackTransaction

This service rolls back an explicit transaction. It must be used in conjunction with a WmART.pub.art.transaction:startTransaction service. If it does not have a corresponding WmART.pub.art.transaction:startTransaction service, your flow service will receive a run-time error.

For more information about implicit and explicit transactions, see "Built-In Services For Explicit Transactions" on page 191.

Input Parameters

rollbackTransactionInput	**Document.** A document that contains the variable *transactionName*, described below.
transactionName	**String.** Used to associate a name with an explicit transaction. The *transactionName* must correspond to the *transactionName* in any WmART.pub.art.transaction:startTransaction or WmART.pub.art.transaction:commitTransaction services associated with the explicit transaction.

WmART.pub.art.transaction:setTransactionTimeout

This service enables you to manually set a transaction timeout interval for implicit and explicit transactions. When you use this service, you are overriding the Integration Server's transaction timeout interval. To change the server's default transaction timeout, see "Changing the Integration Server's Transaction Timeout Interval" on page 194.

You must call this service within a flow before the start of any implicit or explicit transactions. Implicit transactions start when you call an adapter service in a flow. Explicit transactions start when you call the WmART.pub.art.transaction:startTransaction service.

If the execution of a transaction takes longer than the transaction timeout interval, all current executions associated with the flow are cancelled and rolled back if necessary.

This service only overrides the transaction timeout interval for the flow service in which you call it.

Input Parameters

timeoutSeconds	**java.lang.Integer** The number of seconds that the implicit or explicit transaction stays open before the transaction manager aborts it.

WmART.pub.art.transaction:startTransaction

This service starts an explicit transaction. It must be used in conjunction with either a WmART.pub.art.transaction:commitTransaction service or WmART.pub.art.transaction:rollbackTransaction service. If it does not have a corresponding WmART.pub.art.transaction:commitTransaction service or WmART.pub.art.transaction:rollbackTransaction service, your flow service will receive a run-time error.

For more information about implicit and explicit transactions, see "Built-In Services For Explicit Transactions" on page 191.

Input Parameters

startTransactionInput	**Document.** A document that contains the variable *transactionName*, described below.
transactionName	**String.** Used to associate a name with an explicit transaction. The *transactionName* must correspond to the *transactionName* in any WmART.pub.art.transaction:rollbackTransaction or WmART.pub.art.transaction:commitTransaction services associated with the explicit transaction.

Output Parameters

startTransactionOutput	**Document.** A document that contains the variable *transactionName*, described below.
transactionName	**String.** Used to associate a name with an explicit transaction. The *transactionName* must correspond to the *transactionName* in any WmART.pub.art.transaction:rollbackTransaction or WmART.pub.art.transaction:commitTransaction services associated with the explicit transaction.

Changing the Integration Server's Transaction Timeout Interval

Integration Server's default transaction timeout is no timeout (NO_TIMEOUT). To change the server's transaction timeout interval, use a text editor to modify the server.cnf file. Be sure to shut down Integration Server before you edit this file. After you make changes, restart the server.

Add the following parameter to the server.cnf file:

```
watt.art.tmgr.timeout=TransactionTimeout
```

where *TransactionTimeout* is the number of seconds before transaction timeout.

This transaction timeout parameter does not halt the execution of a flow; it is the maximum number of seconds that a transaction can remain open and still be considered valid. For example, if a current transaction has a timeout value of 60 seconds and a flow takes 120 seconds to complete, the transaction manager will rollback all registered operations regardless of the execution status.

For more information about modifying the server.cnf file, see the *webMethods Integration Server Administrator's Guide* for your release.

Transaction Error Situations

When Integration Server encounters a situation that could compromise transactional integrity, it throws an error. Such situations include the following:

- A transaction includes a resource that only supports local transactions.

  If a transaction accesses multiple resources, and more than one of the resources only supports local transactions, the integrity of the transaction cannot be guaranteed. For example, if the first resource successfully commits, and the second resource fails to commit, the first resource interaction cannot be rolled back; it has already been committed. To help prevent this problem, Integration Server detects this case when connecting to more than one resource that does not support two-phase commits. It throws a run-time exception and the service execution fails.

- A transactional or non-transactional resource is used in both a parent transaction and a nested transaction.

  This situation is ambiguous, and most likely means that a nested transaction was not properly closed.

- A parent transaction is closed before its nested transaction.
- After a service request has invoked all its services, but before returning results to the caller, the service may commit its work. This commit could fail if the resource is unavailable or rejects the commit. This will cause the entire server request to fail, and to roll back the transaction.

Specifying Transaction Support in Connections

To support transactions in your adapter, return the appropriate transaction support level in your WmManagedConnectionFactory.queryTransactionSupportLevel implementation.

For local transaction support, override the WmManagedConnection.getLocalTransaction method to return a javax.resource.spi.LocalTransaction object that is capable of interfacing with the transactional capabilities of your resource.

For XA transaction support, override the WmManagedConnection.getXAResource method to return a javax.transaction.xa.XAResource object that is capable of interfacing with the XA transactional capabilities of your resource.

Important:
Do not call the super() method when you override getLocalTransaction or getXAResource.

C Using the Services for Managing Namespace Nodes

- Overview 198
- Connection Services 198
- Adapter Service Services 198
- Listener Services 199
- Listener Notification Services 199
- Polling Notification Services 200

Overview

The ADK provides a set of auxiliary Java services that you can use to replicate namespace nodes programmatically and to change the nodes' metadata appropriately when deploying an adapter to a different Integration Server. It provides services for connections, adapter services, listeners, listener notifications, and polling notifications.

Connection Services

These services are located in the wm.art.dev.connection directory.

Service	Description
wm.art.dev.connection:createConnectionNode	Creates a new connection node in the specified package and folder.
wm.art.dev.connection:deleteConnectionNode	Removes the specified connection node.
wm.art.dev.connection:fetchConnectionManagerMetadata	Returns the connection manager metadata properties that are predefined for all connections.
wm.art.dev.connection:fetchConnectionMetadata	Queries the connection factory and returns the metadata for all properties supported by connections of the specified type.
wm.art.dev.connection:updateConnectionNode	Alters the values of an existing connection.

Adapter Service Services

These services are located in the wm.art.dev.service directory.

Service	Description
wm.art.dev.service:createAdapterServiceNode	Creates a new adapter service node in the specified package and folder from the specified service template and connection alias.
wm.art.dev.service:deleteAdapterServiceNode	Removes a specified adapter service node.

Service	Description
wm.art.dev.service:fetchAdapterServiceTemplateMetadata	Returns all metadata for a specified adapter service template.
wm.art.dev.service:updateAdapterServiceNode	Alters the values of an existing adapter service.

Listener Services

These services are located in the wm.art.dev.listener directory.

Service	Description
wm.art.dev.listener:analyzeListenerNodes	Logs the data for listeners.
wm.art.dev.listener:createListenerNode	Creates a new instance of a listener in the specified package and folder from the specified listener template and connection alias.
wm.art.dev.listener:deleteListenerNode	Removes a specified instance of a listener.
wm.art.dev.listener:fetchListenerTemplateMetadata	Returns all metadata supported by that listener template.
wm.art.dev.listener:updateListenerNode	Alters the values of an existing listener.
wm.art.dev.listener:updateRegisteredNotifications	Checks the registration of listener notifications.

Listener Notification Services

These services are located in the wm.art.dev.notification directory.

Service	Description
wm.art.dev.notification:analyzeListenerNotifications	Logs the data for listener notifications.
wm.art.dev.notification:createListenerNotificationNode	Creates a new instance of a synchronous or asynchronous listener notification in the specified package and folder from the

Service	Description
	specified notification template and connection alias.
wm.art.dev.notification:deleteListenerNotificationNode	Removes a specified instance of listener notification.
wm.art.dev.notification:fetchListenerNotificationTemplateMetadata	Returns all metadata for a specified listener notification template.
wm.art.dev.notification:updateListenerNotificationNode	Alters the values of an existing synchronous or asynchronous listener notification.

Polling Notification Services

These services are located in the wm.art.dev.notification directory.

Service	Description
wm.art.dev.notification:createPollingNotificationNode	Creates a new instance of a polling notification in the specified package and folder from the specified notification template and connection alias.
wm.art.dev.notification:deletePollingNotificationNode	Removes an instance of a specified polling notification.
wm.art.dev.notification:fetchPollingNotificationTemplateMetadata	Returns all metadata for a specified polling notification template, and returns any scheduling properties that are set for the notification.
wm.art.dev.notification:updatePollingNotificationNode	Alters the characteristics of an existing polling notification.

Following are details of each service, presented alphabetically by service name.

wm.art.dev.listener:analyzeListenerNodes

This service logs the data for listeners in the server log file. The data includes the names of associated listener notifications, the class name of listener notifications, their status (active or disabled), and whether the associated listener notification is linked with the same listener or not.

Note:
A listener can be used by multiple listener notifications, but a listener notification will have only one listener node.

Input Parameters

None.

Output Parameters

None.

wm.art.dev.notification:analyzeListenerNotifications

This service logs the data for listener notifications. The data includes the name of a listener to which a listener notification is linked, and the class name of the listener. The service also checks whether the listener notification is registered with the listener or not.

Note:
A listener can be used by multiple listener notifications, but a listener notification will have only one listener node.

Input Parameters

None.

Output Parameters

None.

wm.art.dev.service:createAdapterServiceNode

This service creates a new adapter service node in the specified package and folder from the specified service template and connection alias. You must populate the input pipeline according to the table below before calling this service. To obtain a list of supported adapter service template properties, call the service wm.art.dev.service:fetchAdapterServiceTemplateMetadata.

The value of a property's *systemName* is the internal name of the property. When constructing the input parameter *adapterServiceSettings*, you must use this internal name as the key for setting a property's value. For example, if a service template defines a property named sqlCommand, then its *systemName* (as returned by fetchAdapterServiceTemplateMetadata) would be sqlCommand.If the caller is a Java application, it might then use this information to set this property's value as follows:

```
IData pipeline = IDataFactory.create();
IDataCursor pipeCursor = pipeline.getCursor();
.
.
.
IData svcSettings = IDataFactory.create();
IDataCursor  svcCursor = svcSettings.getCursor();
svcCursor.insertAfter("sqlCommand", "SELECT * FROM fooTable");
.
```

```
.
.
pipeCursor.insertAfter("adapterServiceSettings", svcSettings);
.
.
.
```

This example assumes that the property sqlCommand takes a java.lang.String value.

Use the service's parameters *inputFieldNames, inputFieldTypes, outputFieldNames*,and *outputFieldTypes* to define the properties that comprise the adapter service's input and output signatures. The data types of these properties are arrays of java.lang.String. There is a one-to-one correspondence between the elements in the **FieldNames* and **FieldTypes* arrays. For example, if the property names abc, xyz, and foo are inserted into the *outputFieldNames*parameter, then the service expects that exactly three data types will be inserted into *outputFieldTypes*, and that those data types correspond to the same element in *outputFieldNames*.

Adapter service properties may or may not have default values, depending on the specific adapter's implementation. (In fact, depending on the underlying data type of the property, it might not be possible to assign it a default value.) You may use these default values or override them with values that conform to the underlying data types of the properties. You must explicitly set all required properties in the input parameter *adapterServiceSettings* . A required property is one whose metadata attribute *isRequired*is set to true. (The *isRequired* attribute is contained in fetchAdapterServiceTemplateMetadata.) The absence of the *isRequired*attribute implies that the property is not required. If you fail to set a required property, the service throws an exception. In addition, be aware of any properties that may become required based on the current value of some other property. This service will not try to locate and assign defaults for properties that you have omitted.

Be aware of any resource domains registered by the adapter service template, and set the service's properties according to the interdependencies between resource domains. This includes input and output signatures since they are supported via resource domains. This service provides the properties *inputFieldNames, inputFieldTypes, outputFieldNames*,and *outputFieldTypes*for this purpose. Knowledge of these interdependencies is notification-specific, and beyond the scope of this service. This service does not interpret resource domains.

Input Parameters

serviceName	**String.** Required. The namespace folder:node name of the new adapter service.
packageName	**String.** Required. The package in which to install the adapter service.
connectionAlias	**String.** Required. The namespace folder:node name of the connection.
serviceTemplate	**String.** Required. The fully qualified pathname of the adapter service template class.
adapterServiceSettings	**IData.** Required. The structure for passing the adapter's property values.

.systemName	The value of the *systemName* property; see notes 1, 2, and 3.
.inputFieldNames	**String[].** The names of the fields used in the adapter's input signature.
.inputFieldTypes	**String[].** The data types of the fields used in the adapter's input signature; see note 1.
.outputFieldNames	**String[].** The names of the fields used in the adapter's output signature.
.outputFieldTypes	**String[].** The data types of the fields used in the adapter's output signature; see note 1.

Note:

1. The following Java data types are supported for adapter services: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of all the above data types are also supported.
2. The implementation of the adapter determines whether a particular property is required. To determine whether a property is required, call the service wm.art.dev.service:fetchAdapterServiceTemplateMetadata and check whether its *isRequired* attribute is set to true. If you do not provide a value for a required property, createAdapterServiceNode throws an exception.
3. The number of properties to be configured is adapter-dependent. At a minimum, set those properties whose *isRequired* attribute is set to true.

Output Parameters

None.

wm.art.dev.connection:createConnectionNode

This service creates a new connection node in the specified package and folder. The service initializes the connection in a disabled state. You must populate the input pipeline according to the table below prior to calling this service. To obtain a list of supported connection manager properties and connection-specific properties, call the services wm.art.dev.connection:fetchConnectionManagerMetadata and wm.art.dev.connection:fetchConnectionMetadata.

The value of a property's *systemName* is the internal name of the property. When constructing the input parameters *connectionManagerSettings*and *connectionSettings*, use this internal name as the key for setting a property's value. For example, if a connection defines a property named hostPort, then its *systemName* (as returned by fetchConnectionMetada) would be hostPort. If the caller is a Java application, it might then use this information to set this property's value as follows:

```
IData pipeline = IDataFactory.create();
IDataCursor pipeCursor = pipeline.getCursor();
.
.
.
```

```
IData connSettings = IDataFactory.create();
IDataCursor  connCursor = connSettings.getCursor();
connCursor.insertAfter("hostPort", new Integer(1234));
.
.
.
pipeCursor.insertAfter("connectionSettings", connSettings);
.
.
.
```

This example assumes that the property hostPort takes a java.lang.Integer value. The system names of connection manager properties are predefined. See the output specification of wm.art.dev.connection:fetchConnectionManagerMetadata for their definitions.

All connection manager properties have default values, in the *defaultValue*metadata attribute for each property in the *connectionManagerProperties*structure returned by fetchConnectionManagerMetadata. You may use these default values or override them with values that conform to the underlying data types of the properties. This service will not automatically set property values to their default values; you must explicitly set all required properties in the input parameter *connectionManagerSettings*.A required property is one whose metadata attribute *isRequired*is set to true. (The *isRequired* attribute is contained in fetchConnectionManagerMetada.) The absence of the *isRequired*attribute implies that the property is not required. If you fail to set a required property, the service throws an exception. In addition, be aware of any properties that may become required based on the current value of some other property. This service will not try to locate and assign defaults for properties that you have omitted. In particular, the connection manager property *poolable* is always required. The remaining connection manager properties have a special dependence on the value of *poolable*. Specifically, if you set *poolable* to true, then the remaining connection manager properties must be assigned values as well. If *poolable* is false however, may omit the remaining connection manager properties.

Connection-specific properties may or may not have default values, depending on the specific adapter's implementation. (In fact, depending on the underlying data type of the property, it might not be possible to assign it a default value.) You may use these default values or override them with values that conform to the underlying data types of the properties. This service will not automatically set property values to their default values; you must explicitly set all required properties in the input parameter *connectionSettings.* A required property is one whose metadata attribute *isRequired*is set to true. (The *isRequired* attribute is contained in fetchConnectionMetada.) The absence of the *isRequired*attribute implies that the property is not required. If you fail to set a required property, the service throws an exception. In addition, be aware of any properties that may become required based on the current value of some other property. This service will not try to locate and assign defaults for properties that you have omitted.

Be aware of any resource domains registered by the connection factory, and set the new connection's properties according to the interdependencies between resource domains. Knowledge of these interdependencies is notification-specific, and beyond the scope of this service. This service does not interpret resource domains.

Input Parameters

adapterTypeName	**String**. Required. The name of adapter (the value returned by WmAdapter.getAdapterName()).
connectionFactoryType	**String.** Required. The fully qualified path of the WmManagedConnectionFactory implementation class.
packageName	**String.** Required. The package in which to install the connection.
connectionAlias	**String.** Required. The namespace (folder:node) name of the connection.
connectionManagerSettings	**IData.** Required. The structure for passing connection's manager property values.
.poolable	**Boolean.** Required. Determines whether to pool the connection.
.minimumPoolSize	**Integer.** The minimum number of connections retained in the pool; not used if *.poolable* is false.
.maximumPoolSize	**Integer.** The maximum number of connections retained in the pool; not used if *.poolable* is false.
.poolIncrementSize	**Integer.** The number of connections to add to the pool when additional connections are needed (not to exceed *maximumPoolSize*); not used if *.poolable* is false.
.blockingTimeout	**Integer.** The number of milliseconds to wait for a connection; not used if *.poolable* is false.
.expireTimeout	**Integer.** The number of milliseconds of inactivity that may elapse prior to destroying the connection; not used if *.poolable* is false.
connectionSettings	**IData.** Required. The structure for passing a connection's property values.
.systemName	The value of the *systemName* property; see notes 1, 2, and 3.

Note:

1. Unlike connection manager properties, which are predefined for all connections, actual connection properties (and their underlying data types) vary from adapter to adapter. You must set a connection property's value in accordance with its data type. To determine the value, call the service wm.art.dev.connection:fetchConnectionMetadata and note the *parameterType* attribute for that property.

2. The implementation of the adapter determines whether a particular property is required. To determine whether a property is required, call the service wm.art.dev.connection:fetchConnectionMetadata and check whether its *isRequired* attribute is set to true. If you do not provide a value for a required property, createConnectionNode throws an exception.
3. The number of properties to be configured is adapter-dependent. At a minimum, set those properties whose *isRequired* attribute is set to true

Output Parameters

None.

wm.art.dev.listener:createListenerNode

This service creates a new instance of a listener in the specified package and folder from the specified listener template and connection alias. You must populate the input pipeline according to the table below before calling this service. To obtain a list of supported listener template properties, call wm.art.dev.listener:fetchListenerTemplateMetadata.

The value of a property's *systemName* attribute is the internal name of the property. When constructing the input parameter *listenerSettings*, use this internal name as the key for setting a property's value. For example, if a listener template defines a property called portNumber, then its *systemName (*as returned by fetchListenerTemplateMetadata) would be portNumber.If the caller is a Java application, it might then use this information to set this property's value as follows:

```
IData pipeline = IDataFactory.create();
IDataCursor pipeCursor = pipeline.getCursor();
.
.
.
IData lstnrSettings = IDataFactory.create();
IDataCursor  lstnrCursor = lstnrSettings.getCursor();
lstnrCursor.insertAfter("portNumber", new Integer((int)8888));
.
.
.
pipeCursor.insertAfter("listenerSettings", lstnrSettings);
.
.
.
```

This example assumes that the property *portNumber* takes a java.lang.Integer value.

Listener properties may or may not have default values, depending on the specific listener's implementation. You may use these default values or override them with values that conform to the underlying data types of the properties. You must explicitly set all required properties in the input parameter *listenerSettings.* A required property is one whose metadata attribute *isRequired*is set to true. (The *isRequired* attribute is contained in fetchListenerTemplateMetadata.) The absence of the *isRequired*attribute implies that the property is not required. If you fail to set a required property, the service throws an exception. In addition, be aware of any properties that may become required based on the current value of some other property. This service will not try to locate and assign defaults for properties that you have omitted.

The service initializes the new listener in a disabled state. After calling createListenerNode, you should subsequently call the service to activate it. For more information, see the *webMethods Integration Server Built-In Services Reference*.

You should also consider any resource domains registered by the listener template, and should set the new listener's properties according to the interdependencies between resource domains. Knowledge of these interdependencies is listener-specific, and is beyond the scope of this service. This service does not interpret resource domains.

Input Parameters

listenerName	**String.** Required. The namespace folder:node name of the new listener.
packageName	**String.** Required. The package in which to install the listener.
connectionAlias	**String.** Required. The namespace folder:node name of the connection.
listenerTemplate	**String.** Required. The fully qualified pathname of the listener template class.
listenerSettings	**IData.** Required. The structure for passing the listener's property values.
.systemName	The value of the *systemName* property; see notes 1, 2, and 3.

Note:

1. The following Java data types are supported for listeners: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. There is currently no support for arrays.
2. The implementation of the listener determines whether a particular property is required. To determine whether a property is required, call the service wm.art.dev.listener:fetchListenerTemplateMetadata and check whether its *isRequired* attribute is set to true. If you do not provide a value for a required property, createListenerNode throws an exception.
3. The number of properties to be configured is also listener-dependent. At a minimum, set those properties whose *isRequired* attribute is set to true.

Output Parameters

None.

wm.art.dev.notification:createListenerNotificationNode

This service creates a new instance of a synchronous or asynchronous listener notification in the specified package and folder from the specified notification template and connection alias. You must populate the input pipeline according to the table below before calling this service. To obtain a list of supported listener notification template and schedule properties, call the service wm.art.dev.notification:fetchListenerNotificationTemplateMetadata.

The value of a property's *systemName* attribute is the internal name of the property. When constructing the input parameter *notificationSettings*, use this internal name as the key for setting a property's value. For example, if a notification template defines a property named foo, then its *systemName* (as returned by fetchListenerNotificationTemplateMetadata) would be foo. If a Java application calls the service, it might use this information to set this property's value as follows:

```
IData pipeline = IDataFactory.create();
IDataCursor pipeCursor = pipeline.getCursor();
.
.
.
IData ntfySettings = IDataFactory.create();
IDataCursor  ntfyCursor = ntfySettings.getCursor();
ntfyCursor.insertAfter("foo", "bar");
.
.
.
pipeCursor.insertAfter("notificationSettings",
ntfySettings);
.
.
.
```

This example assumes that the property foo takes a java.lang.String value.

Listener notification properties may or may not have default values, depending on the specific notification's implementation. You may use these default values or override them with values that conform to the underlying data types of the properties. You must explicitly set all required properties in the input parameter *notificationSettings* . A required property is one whose metadata attribute *isRequired*is set to true. (The *isRequired* attribute is contained in fetchListenerNotificationTemplateMetadata.)The absence of the *isRequired*attribute implies that the property is not required. If you fail to set a required property, the service throws an exception. In addition, be aware of any properties that may become required based on the current value of some other property. This service will not try to locate and assign defaults for properties that you have omitted.

The particular combination of input parameters you specify determines whether you create a synchronous or an asynchronous listener notification. The table below identifies which parameters are synchronous- or asynchronous-only. This service throws an exception if you specify invalid or ambiguous combinations of input parameters.

You configure asynchronous listener notifications in a manner similar to how you configure polling notifications. In fact, a polling notification is actually a kind of asynchronous notification. Each time an asynchronous listener notification is triggered, it attempts to generate a run-time publishable output document. The format of this document (the names and types of the data fields it holds)

is predefined. This service provides the input parameter *publishableRecordDef* for this purpose. You construct this parameter in the same manner that you construct *notificationSettings*(as described above). The difference is that the *fieldNames*and *fieldTypes*properties of *publishableRecordDef* each view an array of String. There is a one-to-one correspondence between the elements in these two arrays.The values you assign to these properties should correspond to fields that the notification class outputs in its runNotification()method. (If you specify an empty *publishableRecordDef* it will probably not execute properly.) This service does not validate that the fields in the publishable document are actually generated by the notification. By default, this service creates a publishable document node in a folder named *notificationName*PublishDocument (where *notificationName* is the value you specify for the input parameter *notificationName*). This name is not configurable.

The service initializes the notification in a disabled state. After calling createListenerNotificationNode, you must call the service enableListenerNotification to activate it. For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.

Be aware of any resource domains registered by the listener notification template, and set the new notification's properties according to the interdependencies between resource domains. Knowledge of these interdependencies is notification-specific, and beyond the scope of this service. This service does not interpret resource domains.

Input Parameters

notificationName	**String.** Required. The namespace folder:node name of the new notification.
packageName	**String.** Required. The package in which to install the notification.
listenerNode	**String.** Required. The namespace folder:node name of the notification's listener.
notificationTemplate	**String.** Required. The fully qualified pathname of the listener notification template class; see note 1.
notificationSettings	**IData.** Required. The structure for passing the listener's property values.
.systemName	The value of the *systemName* property; see notes 2, 3, and 4.
serviceName	**String.** For synchronous notifications only; required. The node name of the service to invoke. This parameter must not be null.
publishableRecordDef	**IData**. For asynchronous notifications only; required. The definition of the run-time publishable output document.

.fieldNames	**String[]**. The names of the fields used in the notification's output document.
.fieldTypes	**String[].** The data types of the fields used in the notification's output document; see note 2.
requestRecordDef	**IData**. For synchronous notifications only; required. The definition of the request document sent to *serviceName*. This parameter must not be null, but may be empty.
.fieldNames	**String[]**. The names of fields used in notification's request document
.fieldTypes	**String[]**. The data types of fields used in notification's request document; see note 2.
replyRecordDef	**IData**. For synchronous notifications only; required. The definition of the reply document received from *serviceName*. This parameter must not be null, but may be empty.
.fieldNames	**String[]**. The names of fields used in notification's reply document
.fieldTypes	**String[]**. The data types of fields used in notification's reply document; see note 2.

Note:

1. Synchronous listener notification classes should extend the WmSyncListenerNotification class. Asynchronous listener notifications should extend the WmAsyncListenerNotification class.
2. The following Java data types are supported for listener notifications: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of all the above data types are also supported.
3. The implementation of the notification determines whether a particular property is required. To determine whether a property is required, call the service wm.art.dev.notification:fetchListenerNotificationTemplateMetadata and check whether its *isRequired* attribute is set to true. If you do not provide a value for a required property, createListenerNotificationNode throws an exception.
4. The number of properties to be configured is notification-dependent. At a minimum, set those properties whose *isRequired* attribute is set to true.

Output Parameters

None.

wm.art.dev.notification:createPollingNotificationNode

This service creates a new instance of a polling notification in the specified package and folder from the specified notification template and connection alias. You must populate the input pipeline according to the table below before calling this service. To obtain a list of supported template and schedule properties, call the service wm.art.dev.notification:fetchPollingNotificationTemplateMetadata.

The value of a property's *systemName* attribute is the internal name of the property. When constructing the *notificationSettings*input parameter, use this internal name as the key for setting a property's value. For example, if a notification template defines a property named sqlCommand, then its *systemName* (as returned by fetchPollingNotificationTemplateMetadata) would be sqlCommand. If a Java application calls the service, it might then use this information to set this property's value as follows:

```
IData pipeline = IDataFactory.create();
IDataCursor pipeCursor = pipeline.getCursor();
.
.
.
IData ntfySettings = IDataFactory.create();
IDataCursor  ntfyCursor = ntfySettings.getCursor();
ntfyCursor.insertAfter("sqlCommand", "SELECT * FROM fooTable");
.
.
.
pipeCursor.insertAfter("notificationSettings", ntfySettings);
.
.
.
```

This example assumes that the property sqlCommand takes a java.lang.String value.

Polling notification properties may or may not have default values, depending on the specific notification's implementation. You may use these default values or override them with values that conform to the underlying data types of the properties. You must explicitly set all required properties in the input parameter *notificationSettings*. A required property is one whose metadata attribute *isRequired*is set to true. (The *isRequired* attribute is contained in fetchPollingNotificationTemplateMetadata.)The absence of the *isRequired*attribute implies that the property is not required. If you fail to set a required property, the service throws an exception. In addition, be aware of any properties that may become required based on the current value of some other property. This service will not try to locate and assign defaults for properties that you have omitted.

By default, this service creates a publishable document node in a folder named *notificationName*PublishDocument, where *notificationName* is the value you specify for the input parameter *notificationName.*This name is not configurable.

You specify the format (the names and types of the fields) of the publishable document by populating the *publishableRecordDef*parameter with the names and types of any notification fields that may appear in the document generated by the notification. You construct this parameter in the same manner you construct *notificationSettings*(as described above). The difference is that the *fieldNames*and *fieldTypes*properties of *publishableRecordDef* each view an array of String. There is a

one-to-one correspondence between the elements in these two arrays. Also, the values assigned to these properties should correspond to fields that the notification class outputs in its runNotification()method. (If you specify an empty *publishableRecordDef* it will probably not execute properly.) This service does not validate that the fields in the publishable document are actually generated by the notification.

The service initializes the notification in a disabled state. After calling createPollingNotificationNode, you must call the service enablePollingNotification to activate it. For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.

In addition, you can set the notification's delivery schedule using the input parameter *scheduleSettings*. By default, a newly-created notification has no assigned delivery schedule. You must configure the notification's delivery schedule before calling enablePollingNotification. To do this, provide a valid *scheduleSettings*input parameter to either createPollingNotificationNode or updatePollingNotificationNode.

Be aware of any resource domains registered by the polling notification template, and set the new notification's properties according to the interdependencies between resource domains. Knowledge of these interdependencies is notification-specific, and beyond the scope of this service. This service does not interpret resource domains.

Input Parameters

notificationName	**String.** Required. The namespace folder:node name of the new notification.
packageName	**String.** Required. The package in which to install the notification.
connectionAlias	**String.** Required. The namespace folder:node name of the connection.
notificationTemplate	**String.** Required. The fully qualified pathname of the polling notification template class.
notificationSettings	**IData.** Required. The structure for passing the notification's property values.
.systemName	The value of the *systemName* property; see notes 1, 2, and 3.
scheduleSettings	**IData.** Required. The notification's schedule settings.
.notificationInterval	**Integer**. The frequency of the polling (in seconds).
.notificationOverlap	**Boolean**. Specifies whether notifications can overlap.
.notificationImmediate	**Boolean**. Specifies whether to fire the notification immediately.

publishableRecordDef	**IData.** Required. The definition of the run-time publishable output document.
.fieldName	**String[]**. The names of the fields used in the notification's output document.
.fieldTypes	**String[]**. The data types of the fields used in the notification's output document; see note 1.

Note:

1. The following Java data types are supported for polling notifications: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of all the above data types are also supported.
2. The implementation of the notification determines whether a particular property is required. To determine whether a property is required, call the service wm.art.dev.notification:fetchPollingNotificationTemplateMetadata and check whether its *isRequired* attribute is set to true. If you do not provide a value for a required property, createPollingNotificationNode throws an exception.
3. The number of properties to be configured is notification-dependent. At a minimum, set those properties whose *isRequired* attribute is set to true.

Output Parameters

None.

wm.art.dev.service:deleteAdapterServiceNode

This service removes a specified adapter service node. This action is immediate and non-reversible, and returns no output data. You may assume that the service completed successfully if it does not throw a checked exception.

Input Parameters

serviceName	**String.** Required. The namespace folder:node name of an existing adapter service.

Output Parameters

None.

wm.art.dev.connection:deleteConnectionNode

This service removes the specified connection node. You must disable the connection before you delete it, using the disableConnection service. (For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.) This action is immediate and non-reversible, and

returns no output data. You may assume that the service completed successfully if it does not throw a checked exception.

Input Parameters

connectionAlias	**String.** Required. The namespace folder:node name of the connection.

Output Parameters

None.

wm.art.dev.listener:deleteListenerNode

This service removes a specified instance of a listener. You must disable the listener before deleting it, using the disableListener service. For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.

This action is immediate and non-reversible, and returns no output data. You may assume that the service completed successfully if it does not throw a checked exception.

Input Parameters

listenerName	**String.** Required. The namespace folder:node name of an existing listener.

Output Parameters

None.

wm.art.dev.notification:deleteListenerNotificationNode

This service removes a specified instance of listener notification. You must disable the notification before deleting it, using the disableListenerNotification service. For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.

This action is immediate and non-reversible, and returns no output data. Use the *cascadeDelete* flag to propagate the deletion across the Broker. You may assume that the service completed successfully if it does not throw a checked exception.

Input Parameters

notificationName	**String.** Required. The namespace folder:node name of an existing listener notification.

cascadeDelete	**Boolean**. If true, this parameter propagates the deletion across the Broker.

Output Parameters

None.

wm.art.dev.notification:deletePollingNotificationNode

This service removes an instance of a specified polling notification. You must disable the notification before deleting it, using the disablePollingNotificationNode service. For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.

This action is immediate and non-reversible, and returns no output data. Use the *cascadeDelete* flag to propagate the deletion across Broker. You may assume that the service completed successfully if it does not throw a checked exception.

Input Parameters

notificationName	**String.** Required. The namespace folder:node name of an existing polling notification.
cascadeDelete	**Boolean**. If true, this parameter propagates the deletion across Broker.

Output Parameters

None.

wm.art.dev.service:fetchAdapterServiceTemplateMetadata

Given a specified connection alias and adapter service template class path, this service returns all metadata for an adapter service template.

The service returns a *templateProperties*array containing all metadata associated with each of the adapter service template properties. Of particular interest are the *systemName, parameterType, defaultValue,*and *isRequired*attributes, which you can use to configure a new adapter service node.

Input Parameters

connectionAlias	**String.** Required. The namespace folder:node name of the connection.
serviceTemplate	**String.** Required. The fully qualified pathname of the adapter service template class.

Output Parameters

description	**String.** Required. The value returned by WmDescriptor.getDescription().
displayName	**String.** Required. The value returned by WmDescriptor.getDisplayName().
templateURL	**String.** Required. The URL of the online help page for the adapter service.
indexMaps[i]	**IData[].** Required. An *i*-dimensioned array of field maps.
.mapName	**String**. The field map name.
.isVariable	**Boolean**. Specifies whether the field map is variable length.
templateProperties[n]	**IData**. Required. An *n*-dimensioned array of properties.
.systemName	**String.** Required. The internal property name.
.displayName	**String.** Required. The external displayable property name.
.description	**String.** Required. The Property description.
.parameterType	**String.** Required. The data type of property; see note 1.
.groupURL	**String.** The URL of the group's help page.
.groupName	**String.** The name of the group to which the property belongs.
.tupleName	**String.** The name of the tuple to which the property belongs.
.treeName	**String.** The name of the tree to which the property belongs.
.treeDelimiter	**String.** The delimiter character used in the tree.
.resourceDomain	**String.** The resource domain name and dependencies.
.defaultValue	**String.** The default property value.
.isRequired	**Boolean.** Specifies whether the property is required.
.isHidden	**Boolean.** Specifies whether the property is displayable.
.isReadOnly	**Boolean.** Specifies whether the property can be modified.
.isFill	**Boolean.** Specifies whether the editors should pre-fill the property.
.isPassword	**Boolean.** Specifies whether the property is a password.

.isMultiline	**Boolean.** Specifies whether the property traverses multiple lines.
.isKey	**Boolean.** Specifies whether the property is a key field.
.minSeqLength	**String**. The lower bound of the sequence.
.maxSeqLength	**String**. The upper bound of the sequence.
.minStringLength	**String**. The minimum string length.
.maxStringLength	**String**. The maximum string length.
.useParam	**String**. Specifies whether the property is available for use.

Note:

1. The following Java data types are supported for adapter services: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of all the above data types are also supported.

wm.art.dev.connection:fetchConnectionManagerMetadata

This service returns the connection manager metadata properties that are predefined for all connections, such as *poolable*, *minimumPoolSize*, *maximumPoolSize*, and others. (These are the properties of *connectionManagerSettings*.) No inputs are required to run this service.

The service returns a *connectionManagerProperties* array containing all metadata associated with each of the connection manager properties. Of particular interest are the *systemName, parameterType, defaultValue,* and *isRequired*attributes, which you can use to configure a connection node.

Input Parameters

None.

Output Parameters

connectionManagerProperties[n]	**IData**. Required. An *n*-dimensioned array of connection manager properties.
.systemName	**String**. Required. The internal property name, which is one of the following: poolable, minimumPoolSize, maximumPoolSize, poolIncrementSize, blockingTimeout, or expireTimeout.
.displayName	**String**. Required. The external displayable property name.

.description	**String**. Required. The displayable property description.
.parameterType	**String**. Required. The data type of property; see note 1.
.groupURL	**String**. Not applicable to connection manager properties.
.groupName	**String**. Not applicable to connection manager properties.
.tupleName	**String**. Not applicable to connection manager properties.
.treeName	**String**. Not applicable to connection manager properties.
.treeDelimiter	**String**. Not applicable to connection manager properties.
.resourceDomain	**String**. Not applicable to connection manager properties.
.defaultValue	**String**. The default property value.
.isRequired	**Boolean**. Specifies whether the property is required.

Note:
The following Java data types are supported for connections: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays are not currently supported.

wm.art.dev.connection:fetchConnectionMetadata

Given an adapter type name and a fully qualified connection factory class path, this service queries the connection factory and returns the metadata for all properties supported by connections of that type.

For example, a Java client might invoke this service as follows:

```
IData pipeline = IDataFactory.create();
IDataCursor pipeCursor = pipeline.getCursor();
pipeCursor.insertAfter("adapterTypeName",
"FooAdapter");
pipeCursor.insertAfter("connectionFactoryType",
                       "com.wm.adapters.FooConnFactory");
ExtendedConnectionUtils.fetchConnectionMetadata(pipeline);
.
.
.
```

The service returns a *connectionProperties*array containing all metadata associated with each of the connection properties. Of particular interest are the *systemName, parameterType, defaultValue,*and *isRequired*attributes, which you can use to configure a connection node.

Input Parameters

adapterTypeName	**String**. Required. The name of adapter (the value returned by WmAdapter.getAdapterName()).
connectionFactoryType	**String.** Required. The fully qualified path of the WmManagedConnectionFactory implementation class.

Output Parameters

description	**String.** Required. The value returned by WmDescriptor.getDescription().
displayName	**String.** Required. The value returned by WmDescriptor.getDisplayName().
templateURL	**String**. Required. The URL of online help page for the connection.
connectionProperties[n]	**IData[].** Required. An *n*-dimensioned array of properties.
.systemName	**String**. Required. The internal property name (adapter-specific).
.displayName	**String**. Required. The external displayable property name.
.description	**String**. Required. The displayable property description.
.parameterType	**String**. Required. The data type of property; see note 1.
.groupURL	**String**. The URL of the group's help page.
.groupName	**String**. The name of the group to which the property belongs.
.tupleName	**String**. The name of the tuple to which the property belongs.
.treeName	**String**. The name of the tree to which the property belongs.

.treeDelimiter	**String**. The delimiter character used in the tree.
.resourceDomain	**String**. The resource domain name.
.defaultValue	**String**. The default property value.
.isRequired	**Boolean**. Specifies whether the property is required.
.isHidden	**Boolean**. Specifies whether the property is displayable.
.isReadOnly	**Boolean**. Specifies whether the property can be modified.
.isFill	**Boolean**. Specifies whether the editors should pre-fill the property.
.isPassword	**Boolean**. Specifies whether the property is a password.
.isMultiline	**Boolean**. Specifies whether the property traverses multiple lines.
.isKey	**Boolean**. Specifies whether the property is a key field.
.minSeqLength	**String**. The lower bound of the sequence.
.maxSeqLength	**String**. The upper bound of the sequence.
.minStringLength	**String**. The minimum string length.
.maxStringLength	**String**. The maximum string length.
.useParam	**String**. Specifies whether the property is available for use.

Note:
The following Java data types are supported for connections: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays are not currently supported.

wm.art.dev.listener:fetchListenerTemplateMetadata

Given a valid connection alias and listener template class path, this service returns all metadata supported by that listener template.

The service returns a *templateProperties*array containing all metadata associated with each of the listener template properties. Of particular interest are the *systemName, parameterType, defaultValue,*and *isRequired*attributes, which you can use to configure a new listener instance.

Input Parameters

connectionAlias	**String**. Required. The namespace folder:node name of the connection.
listenerTemplate	**String.** Required. The fully qualified pathname of listener template class.

Output Parameters

description	**String**. Required. The value returned by WmDescriptor.getDescription().
displayName	**String**. Required. The value returned by WmDescriptor.getDisplayName().
templateURL	**String**. Required. The URL of the online help page for the adapter service.
indexMaps[i]	**IData[]**. Required. An *i*-dimensioned array of field maps.
.mapName	**String**. The field map name.
.isVariable	**Boolean**. Specifies whether the field map is variable length.
templateProperties[n]	**IData[]**. An *n*-dimensioned array of properties.
.systemName	**String**. Required. The internal property name.
.displayName	**String**. The external displayable property name.
.description	**String**. The property description.
.parameterType	**String**. Required. The data type of the property; see note 1.
.groupURL	**String**. The URL of the group's help page.
.groupName	**String**. The name of the group to which the property belongs.
.tupleName	**String**. The name of the tuple to which the property belongs.
.treeName	**String**. The name of the tree to which the property belongs.

.treeDelimiter	**String**. The delimiter character used in the tree.
.resourceDomain	**String**. The resource domain name and dependencies.
.defaultValue	**String**. The default property value.
.isRequired	**Boolean**. Specifies whether the property is required.
.isHidden	**Boolean**. Specifies whether the property is displayable.
.isReadOnly	**Boolean**. Specifies whether the property can be modified.
.isFill	**Boolean**. Specifies whether the editors should pre-fill the property.
.isPassword	**Boolean**. Specifies whether the property is a password.
.isMultiline	**Boolean**. Specifies whether the property traverses multiple lines.
.isKey	**Boolean**. Specifies whether the property is a key field.
.minSeqLength	**String**. The lower bound of the sequence.
.maxSeqLength	**String**. The maximum string length.
.useParam	**String**. Specifies whether the property is available for use.

Note:
The following Java data types are supported for listeners: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of the above data types are not supported.

wm.art.dev.notification:fetchListenerNotificationTemplateMetadata

Given a valid listener node name and listener notification template class path, this service returns all metadata for that listener notification template.

This service returns a *templateProperties*array containing all metadata associated with each notification template property. Of particular interest are the *systemName, parameterType, defaultValue,*and *isRequired*attributes, which you can use to configure a new listener notification instance.

Input Parameters

listenerNode	**String**. Required. The namespace folder:node name of a configured listener.
notificationTemplate	**String.** Required. The fully qualified pathname of polling notification template class.

Output Parameters

description	**String**. Required. The value returned by WmDescriptor.getDescription().
displayName	**String**. Required. The value returned by WmDescriptor.getDisplayName().
templateURL	**String**. Required. The URL of online help page for the adapter service.
indexMaps[i]	**IData[]**. Required. An *i*-dimensioned array of field maps.
.mapName	**String**. The field map name.
.isVariable	**Boolean**. The field map is variable length.
templateProperties[n]	**IData[]**. Required. An *n*-dimensioned array of properties.
.systemName	**String**. Required. The internal property name.
.displayName	**String.** The external displayable property name.
.description	**String**. The property description.
.parameterType	**String**. Required. The data type of property; see note 1.
.groupURL	**String**. The URL of the group's help page.
.groupName	**String**. The name of the group to which the property belongs.
.tupleName	**String**. The name of the tuple to which the property belongs.
.treeName	**String**. The name of the tree to which the property belongs.
.treeDelimiter	**String**. The delimiter character used in the tree.

.resourceDomain	**String**. The resource domain name and dependencies.
.defaultValue	**String**. The default property value.
.isRequired	**Boolean**. Specifies whether the property is required.
.isHidden	**Boolean**. Specifies whether the property is displayable.
.isReadOnly	**Boolean**. Specifies whether the property can be modified.
.isFill	**Boolean**. Specifies whether the editors should pre-fill the property.
.isPassword	**Boolean**. Specifies whether the property is a password.
.isMultiline	**Boolean**. Specifies whether the property traverses multiple lines.
.isKey	**Boolean**. Specifies whether the property is a key field.
.minSeqLength	**String**. The lower bound of the sequence.
.maxSeqLength	**String**. The upper bound of the sequence.
.minStringLength	**String**. The minimum string length.
.maxStringLength	**String**. The maximum string length.
.useParam	**String**. Specifies whether the property is available for use.

Note:
The following Java data types are supported for listener notifications: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of all the above data types are also supported.

wm.art.dev.notification:fetchPollingNotificationTemplateMetadata

Given a valid connection alias and polling notification template class path, this service returns all metadata for that polling notification template. It also returns any scheduling properties that are set for the notification.

This service returns a *templateProperties*array containing all metadata associated with each notification template property. The *scheduleProperties*array contains the names of the properties used to configure a notification's schedule. Of particular interest are the *systemName*, *parameterType*,

*defaultValue,*and *isRequired*attributes, which you can use to configure a new polling notification instance.

Input Parameters

connectionAlias	**String**. Required. The namespace folder:node name of the connection.
notificationTemplate	**String.** Required. The fully qualified pathname of the polling notification template class.

Output Parameters

description	**String**. Required. The value returned by WmDescriptor.getDescription().
displayName	**String**. Required. The value returned by WmDescriptor.getDisplayName().
templateURL	**String**. Required. The URL of the online help page for the adapter service.
indexMaps[i]	**IData[]**. Required. An *i*-dimensioned array of field maps.
.mapName	**String**. The field map name.
.isVariable	**Boolean**. Specifies whether the field map is variable length.
templateProperties[n]	**IData[]**. Required. An *n*-dimensioned array of properties.
.systemName	**String**. Required. The internal property name.
.displayName	**String**. The external displayable property name.
.description	**String**. The property description.
.parameterType	**String**. Required. The data type of the property; see note 1.
.groupURL	**String**. The URL of the group's help page.
.groupName	**String**. The name of the group to which the property belongs.
.tupleName	**String**. The name of the tuple to which the property belongs.
.treeName	**String**. The name of the tree to which the property belongs.
.treeDelimiter	**String**. The delimiter character used in the tree.
.resourceDomain	**String**. The resource domain name and dependencies.
.defaultValue	**String**. The default property value.

.isRequired	**Boolean**. Specifies whether the property is this a required.
.isHidden	**Boolean**. Specifies whether the property is displayable.
.isReadOnly	**Boolean**. Specifies whether the property can be modified.
.isFill	**Boolean**. Specifies whether the editors should pre-fill the property.
.isPassword	**Boolean**. Specifies whether the property is a password.
.isMultiline	**Boolean**. Specifies whether the property traverses multiple lines.
.isKey	**Boolean**. Specifies whether the property is a key field.
.minSeqLength	**String**. The lower bound of the sequence.
.maxSeqLength	**String**. The upper bound of the sequence.
.minStringLength	**String**. The minimum string length.
.maxStringLength	**String**. The maximum string length.
.useParam	**String**. Specifies whether the property is available for use.
scheduleProperties[j]	**IData[]**. A *j*-dimensioned array of scheduling properties.
.systemName	**String**. Required. The internal property name
.parameterType	**String**. Required. The data type of scheduling property

Note:
The following Java data types are supported for polling notifications: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of all the above data types are also supported.

wm.art.dev.service:updateAdapterServiceNode

This service alters the values of an existing adapter service. You must populate the input pipeline according to the table below before calling this service. To obtain a list of supported template and schedule properties, call the service wm.art.dev.service:fetchAdapterServiceTemplateMetadata.

The value of the *systemName* metadata attribute is the internal name of a property. When constructing the input parameter *adapterServiceSettings*, use this internal name as the key for setting a property's value. For an example of setting an adapter service property, see the Java code example in wm.art.dev.service:createAdapterServiceNode.

You only need to supply values for the properties you want to change. This service attempts to overlay these new values on the adapter service's current property values. The resulting set of merged property values are used to reconfigure the adapter service.

You can change the connection resource that the adapter service uses by providing a new *connectionAlias*input parameter. If you omit this parameter, the adapter service will continue to use its current connection resource. If you are changing only the connection resource, it is not necessary to provide the *adapterServiceSettings*input parameter.

When providing explicit property values in *adapterServiceSettings*, you must provide values that conform to the underlying data types of those properties.

Be aware of any resource domains registered by the adapter service, and set the service's properties according to the interdependencies between resource domains. This includes input and output signatures since they are supported via resource domains. The properties *inputFieldNames*, *inputFieldTypes*, *outputFieldNames*, and *outputFieldTypes*are provided for this purpose. Knowledge of these interdependencies is adapter-specific, and beyond the scope of this service. This service does not interpret resource domains.

Input Parameters

serviceName	**String**. Required. The namespace folder:node name of an existing adapter service.
connectionAlias	**String**. The namespace folder:node name of the connection to use.
adapterServiceSettings	**IData**. The structure for passing the adapter's property values.
.systemName	The value of the *systemName* property; see notes 1 and 2.
.inputFieldNames	**String[]**. The names of the fields used in the adapter's input signature.
.inputFieldTypes	**String[]**. The data types of the fields used in the adapter's input signature; see note 1.
.outputFieldNames	**String[]**. The names of the fields used in the adapter's output signature.
.outputFieldTypes	**String[]**. The data types of the fields used in the adapter's output signature; see note 1.

Note:

1. The following Java data types are supported for adapter services: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of all the above data types are also supported.
2. Set only those properties that you want to change. If a property's data type is non-primitive (that is, derived from java.lang.Object), you may clear a property's current value by setting it to null. There is no way to do this for Java primitives; they may only be redefined.

Output Parameters

None.

wm.art.dev.connection:updateConnectionNode

This service alters the values of an existing connection. You must disable the connection before attempting this operation, using the disableConnection service. For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.

You must populate the input pipeline according to the table below before calling this service. To obtain a list of supported template and schedule properties, call the services wm.art.dev.connection:fetchConnectionMetadata and wm.art.dev.connection:fetchConnectionManagerMetadata.

The value of the *systemName* metadata attribute is the internal name of a property. When constructing the input parameters *connectionManagerSettings*and *connectionSettings*, use this internal name as the key for setting a property's value. For an example of setting a connection property, see the Java code example in wm.art.dev.connection:createConnectionNode.

The system names of connection manager properties are predefined. See the output specification of wm.art.dev.connection:fetchConnectionManagerMetadata for their definitions.

You only need to supply values for the properties you want to change. This service attempts to overlay these new values on the connection's current property values. The resulting set of merged property values are used to reconfigure the connection. If you are not changing any connection manager or connection-specific properties, it is not necessary to pass in that container parameter. For instance, if you are not changing any connection manager properties, you do not need to build and pass in the *connectionManagerSettings*parameter.

When providing explicit property values in *connectionManagerSettings*and *connectionSettings*, you must provide values that conform to the underlying data types of those properties.

Input Parameters

connectionAlias	**String**. Required. The namespace (folder:node) name of the connection.
connectionManagerSettings	**IData**. The structure for passing the connection's manager property values.
.poolable	**Boolean**. Specifies whether to pool the connection.
.minimumPoolSize	**Integer**. The minimum number of connections retained in the pool; not used if *.poolable* is false.
.maximumPoolSize	**Integer**. The maximum number of connections retained in pool; not used if *.poolable* is false.

.poolIncrementSize	**Integer**. The number of connections to add to the pool when additional connections are needed (not to exceed *maximumPoolSize*). Not used if *.poolable* is false.
.blockingTimeout	**Integer**. The number of milliseconds to wait for a connection; not used if *.poolable* is false.
.expireTimeout	**Integer**. The number of milliseconds of inactivity that may elapse prior to destroying the connection; not used if *.poolable* is false.
connectionSettings	**IData**. The structure for passing the connection's property values.
.systemName	The value of the *systemName* property; see notes 1 and 2.

Note:

1. Unlike connection manager properties, which are predefined for all connections, actual connection properties (and their underlying data types) vary from adapter to adapter. You should set a connection property's value in accordance with its data type. To determine the value, call the service wm.art.dev.connection:fetchConnectionMetadata and note the *parameterType* attribute for that property.
2. Set only those properties that you want to change. If a property's data type is non-primitive (that is, derived from java.lang.Object) you may "undefined" its current value by setting it to null. There is no way to do this for Java primitives; they may only be redefined.

Output Parameters

None.

wm.art.dev.listener:updateListenerNode

This service alters the values of an existing listener. You must populate the input pipeline according to the table below before calling this service. To obtain a list of supported template and schedule properties, call the service wm.art.dev.listener:fetchListenerTemplateMetadata. You must disable a listener before updating its properties, using the disableListener service. For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.

The value of the *systemName* metadata attribute is the internal name of a property. When constructing the input parameter *listenerSettings*, use this internal name as the key for setting a property's value. For an example of setting a connection property, see the Java code example in wm.art.dev.listener:createListenerNode.

You only need to supply values for the properties you want to change. This service will attempt to overlay these new values on the listener's current property values. The resulting set of merged property values will be used to reconfigure the listener.

You can change the connection resource that the listener uses by providing a new *connectionAlias*input parameter. If you omit this parameter, the listener will continue to use its current connection resource. If you are changing only the connection resource, it is not necessary to provide the *listenerSettings*input parameter.

When providing explicit property values in *listenerSettings*, you must provide values that conform to the underlying data types of those properties.

Be aware of any resource domains registered by the listener, and set the listener's properties according to the interdependencies between resource domains. Knowledge of these interdependencies is adapter-specific, and beyond the scope of this service. This service does not interpret resource domains.

Input Parameters

listenerName	**String**. Required. The namespace folder:node name of an existing listener.
connectionAlias	**String**. The namespace folder:node name of the connection to use.
listenerSettings	**IData**. The structure for passing listener property values.
.systemName	The value of the *systemName* property; see notes 1 and 2.

Note:

1. The following Java data types are supported for adapter services: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays are not supported.
2. Set only those properties that you want to change. If a property's data type is non-primitive (that is, derived from java.lang.Object) you may "undefined" its current value by setting it to null. There is no way to do this for Java primitives; they may only be redefined.

Output Parameters

None.

wm.art.dev.notification:updateListenerNotificationNode

This service alters the values of an existing synchronous or asynchronous listener notification. You must populate the input pipeline according to the table below before calling this service. To obtain a list of supported template and schedule properties, call the service wm.art.dev.notification:fetchListenerNotificationTemplateMetadata.You must disable a listener notification before updating its properties, using the disableListenerNotification service. For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.

You should be familiar with the differences between synchronous and asynchronous listener notifications, as described in wm.art.dev.notification:createListenerNotificationNode.

Once you configure a synchronous notification, you cannot convert it to an asynchronous one. Nor can you convert an asynchronous notification to a synchronous one.

In the table below, two input parameters apply to both synchronous and asynchronous listener notifications: *listenerNode*and *notificationSettings*.For each of these parameters, the service attempts to overlay the given values onto the notification's current values. The resulting set of merged property values are used to reconfigure the notification.

For instance, you can change the listener resource used by the notification by providing a new *listenerNode*input parameter. If you omit this parameter, the notification continues to use its current listener. If you are changing only the listener resource, it is not necessary to provide the *notificationSettings* or any other input parameters.

For an example of setting values within the pipeline, see wm.art.dev.notification:createListenerNotificationNode. In particular, note that when constructing the *notificationSettings*parameter, the value of the *systemName* metadata attribute is an adapter-specific internal name of a property. You should substitute the actual internal system property name for the *systemName* placeholder. (You can fetch the name using wm.art.dev.notification:fetchListenerNotificationTemplateMetadata.)

For synchronous notifications there are three additional input parameters that may be provided: *serviceName, requestRecordDef,*and *replyRecordDef.*The first one specifies the node name of a separate service to be invoked by the notification at run time. If you omit this parameter, the notifications still retains the current value of the parameter. The latter two parameters view document record definitions that the notification uses to format messages when communicating with this service. You can modify these definitions by providing *requestRecordDef*or *replyRecordDe f* (or both). You populate these parameters in the same manner that you populate *notificationSettings*(as described above). The difference is that the *fieldNames*and *fieldTypes*properties each view an array of String. There is a one-to-one correspondence between the elements in these two arrays.The values assigned to these properties should correspond to fields that the notification class and the invoked service will actually use when communicating with each other. Note that this service has no way to verify this assertion. If you mis-configure either record definition, the consequence may not be manifested until run time.

You can use the input parameter *publishableRecordDef* to redefine the structure of an asynchronous notification's output document. You configure this parameter in the same way that you configure *requestRecordDef*and *replyRecordDef* for synchronous notifications. The same caveats apply as well: you are responsible for knowing how to define this document such that it conforms to the notification's expectations.

Unlike *notificationSettings, listenerNode,*and *serviceName,*the parameters *requestRecordDef, replyRecordDef,*and *publishableRecordDef*are used to replace their existing counterparts in their entirety. The service does not attempt to merge the values in these parameters with the notification's current values. For instance, if you want to redefine a synchronous notification's reply record definition, you must define the entire record in the *replyRecordDef*input parameter.

When specifying property valuesyou must provide values that conform to the underlying data types of those properties.

Be aware of any resource domains registered by the notification, and set the notification's properties according to the interdependencies between resource domains. Knowledge of these interdependencies is adapter-specific, and beyond the scope of this service. This service does not interpret resource domains.

Input Parameters

notificationName	**String**. Required. The namespace folder:node name of an existing polling notification.
listenerNode	**String**. The namespace folder:node name of the associated listener.
notificationSettings	**IData**. The structure for passing notification property values.
.systemName	The value of the *systemName* property; see notes 1 and 2.
serviceName	**String**. The node name of the service to invoke (synchronous only).
publishableRecordDef	**IData**. The output document definition (asynchronous only).
.fieldNames	**String[]**. The names of fields used in notification's output document; see note 3.
.fieldTypes	**String[]**. The data types of fields used in notification's output document; see notes 1 and 3.
requestRecordDef	**IData**. The definition of the request document sent to *serviceName*(synchronous only).
.fieldNames	**String[]**. The names of fields used in the request document; see note 3.
.fieldTypes	**String[]**. The data types of fields used in the request document; see notes 1 and 3.
replyRecordDef	**IData**. The definition of the reply document received from *serviceName*(synchronous only).
.fieldNames	**String[]**. The names of the fields used in the reply document; see note 3.
.fieldTypes	**String[]**. The data types of the fields used in the reply document; see notes 1 and 3.

Note:

1. The following Java data types are supported for adapter services: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer,

java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of all the above data types are also supported.

2. Set only those properties that you want to change. If a property's data type is non-primitive (that is, derived from java.lang.Object) you may clear a property's current value by setting it to null. There is no way to do this for Java primitives; they may only be redefined.
3. Values inserted into fieldNames must correspond in number and order to the values inserted into fieldTypes. Additionally, you must provide values defining the entire document; the service does not attempt to merge these values into the current document definition.

Output Parameters

None.

wm.art.dev.notification:updatePollingNotificationNode

This service alters the characteristics of an existing polling notification. You must populate the input pipeline according to the table below before calling this service. To obtain a list of supported template and schedule properties, call the service wm.art.dev.notification:fetchPollingNotificationTemplateMetadata. You must disable a polling notification before updating its properties, using the disablePollingNotification service. For more information, see the *webMethods Integration Server Built-In Services Reference* for your release.

You might want to modify the following items:

- The underlying connection resource used by the notification
- Its metadata property values
- Its schedule
- The signature of its publishable output document

These items are represented, respectively in the input pipeline as *connectionAlias*, *notificationSettings*, *scheduleSettings*,and *publishableRecordDef*.You may provide values in any or all of these parameters when calling this service. For the first three of these parameters, the service attempts to overlay the given values onto the notification's current values. The resulting set of merged property values are used to reconfigure the polling notification.

For instance, you can change the connection resource that the polling notification uses by providing a new *connectionAlias*input parameter. If you omit this parameter, the notification continues to use its current connection resource. If you are changing only the connection resource, it is not necessary to provide the *notificationSettings*,*scheduleSettings*,or *publishableRecordDef*input parameters.

For an example of setting values within the pipeline, see wm.art.dev.notification:createPollingNotificationNode. In particular, note that when constructing the *notificationSettings*parameter, the value of the *systemName* metadata attribute is the adapter-specific internal name of a property. You should substitute the actual internal system property name for the *systemName* placeholder. (You can fetch the name using wm.art.dev.notification:fetchPollingNotificationTemplateMetadata.)

Unlike *notificationSettings*and *scheduleSettings*, with *publishableRecordDef*you must provide values defining the entire document; the service does not attempt to merge these values into the current

publishable record definition. The values in *publishableRecordDef* will replace the current definition values. You populate the *publishableRecordDef* parameter in much the same manner that you populate *notificationSettings* (as described above). The difference is that the *fieldNames* and *fieldTypes* properties view an array of String. There is a one-to-one correspondence between the elements in these two arrays.

When specifying property values, you must provide values that conform to the underlying data types of those properties.

Be aware of any resource domains registered by the notification, and set the notification's properties according to the interdependencies between resource domains. Knowledge of these interdependencies is adapter-specific, and beyond the scope of this service. This service does not interpret resource domains.

Input Parameters

notificationName	**String**. Required. The namespace folder:node name of an existing polling notification.
connectionAlias	**String**. The namespace folder:node name of the connection to use.
notificationSettings	**IData**. The structure for passing notification property values.
.systemName	The value of the *systemName* property; see notes 1 and 2.
scheduleSettings	**IData**. The notification's schedule settings.
.notificationInterval	**Integer**. The polling frequency; see note 2.
.notificationOverlap	**Boolean**. Specifies whether the notifications can overlap; see note 2.
.notificationImmediate	**Boolean**. Specifies whether to fire the notification immediately; see note 2.
publishableRecordDef	**IData**. The output document definition.
.fieldNames	**String[]**. The names of the fields used in the notification's output document; see note 3.
.fieldTypes	**String[]**. The data types of the fields used in the notification's output document; see notes 1 and 3.

Note:

1. The following Java data types are supported for adapter services: char, short, int, long, float, double, boolean, java.lang.String, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Boolean. Arrays of all the above data types are also supported.

2. Set only those properties that you want to change. If a property's data type is non-primitive (that is, derived from java.lang.Object) you may "undefined" its current value by setting it to null. There is no way to do this for Java primitives; they may only be redefined.
3. Values inserted into fieldNames must correspond in number and order to the values inserted into fieldTypes. Additionally, you must provide values defining the entire document; the service does not attempt to merge these values into the current publishable record definition.

Output Parameters

None.

wm.art.dev.listener:updateRegisteredNotifications

This service checks listener notifications, finds the linked listener, and then checks whether the listener notification is also registered for the same listener or not. If not, then the service registers the listener notification with the linked listener.

Note:
A listener can be used by multiple listener notifications, but a listener notification will have only one listener node. In an ideal case, if a listener notification is linked with a listener, then the listener user interface must show the entry of the listener notification.

Input Parameters

listenerNodeName	**String.** Specifies the name of the WmART-based listener for which notifications will be updated. ■ To update notifications for a specific listener, specify its name. ■ To update notifications for all ART listeners, specify *.

Output Parameters

None.

D Using the Sample Adapter

- Overview 238
- The Sample Server 239
- Banking Functions 240
- Banking Event Queries 240
- Banking Alerts 241
- Prerequisites for Code Compilation 241
- Phase 1: Creating an Adapter Definition 242
- Phase 2: Adding a Connection 244
- Phase 3: Adding Adapter Services 249
- Phase 4: Adding Polling Notifications 255
- Phase 5: Adding Listener Notifications 263

Overview

The ADK provides an example adapter package named WmSampleAdapter. You can use this adapter as a model for developing your own adapters. This adapter enables you to exchange data with a simulated back-end resource provided with the ADK, named Sample Server. You will configure this adapter to perform a banking application. All of the underlying Sample Adapter class files are located in the *Integration Server_directory* \packages\WmSampleAdapter directory. This appendix describes how the Sample Adapter was developed, and how you can configure and run it. The adapter is provided in five phases. Each phase is a standalone adapter, delivered in its own source files; you can build, configure, and run each phase separately. Each phase includes new functionality such that the first phase consists of just the basic framework of the adapter, and the final phase is the fully functional adapter. These phases are as follows:

- **Phase 1**: Creating the adapter definition.

  An adapter definition is recognized as an adapter by webMethods Integration Server, but it lacks functionality. In the next four phases of development, the adapter includes new functionality including an adapter connection, an adapter service template, a polling notification template, and a listener notification template.

- **Phase 2**: Adding a connection that connects clients to the Sample Server.

- **Phase 3**: Adding adapter services.

  You will create adapter services that perform operations such as depositing, clearing, and bouncing checks, withdrawing and transferring funds, and other operations, as described in "Banking Functions" on page 240.

- **Phase 4**: Adding polling notifications.

  You will configure polling notification nodes that query the Sample Server and publish documents when the following events occur:

  - Using the Clear Check service or the Bounce Check service clears (approves) or bounces (disapproves) a deposited check.
  - Using the Withdraw service or the Transfer service causes a negative account balance.

- **Phase 5**: Adding listener notifications.

  This phase represents the adapter with all its functionality. You will configure a listener to monitor the Sample Server for "alerts" generated by the Sample Server. An alert is a Sample Server mechanism that informs the adapter that an event has occurred. You will configure the listener notification nodes to publish notification documents when the Sample Server generates the following alerts:

  - The Deposit service successfully deposits a check. (You can then invoke the Clear Check service or the Bounce Check service to approve or disapprove the check.)
  - The Withdraw service or the Transfer service causes a negative account balance.

The Sample Server

The back-end resource, Sample Server, is delivered with the ADK as a Java executable. It simulates a banking system. The Sample Server and the adapter communicate with each other using the TCP/IP protocol. You should run the Sample Server as a separate process on any networked computer.

The Sample Adapter uses a document-based messaging scheme to exchange data with the Sample Server. SampleClient.jar, which is also included with the ADK, provides the classes necessary to implement these messages. These messages provide access to a set of business functions, queries, and alerts that are supported by the Sample Server. There is also a metadata repository lookup feature that allows the adapter to retrieve a list of the available functions, queries, and alerts, and to obtain details of the data fields used in each one.

Sample Server Client APIs

WmSampleConnection communicates with the Sample Server using the Sample Server client API. Javadocs for the API are available in the following directory:

Integration Server_directory \packages\WmSampleAdapter\backendResource\doc

where *Integration Server_directory* is the directory in which you installed Integration Server.

Transaction Control

You can configure the client connection for either no transaction control (auto commit mode) or local transaction control across multiple service invocations. The SampleAdapter allows the adapter user to select the transaction type for each WmSampleConnection node.

Customizing the Behavior of the Sample Server

You can control the behavior of the Sample Server using a property file, SampleServer.properties, located in the startup directory of the Sample Server. Detailed comments are embedded among the properties defined in the file. You can edit the file to suit your needs, and then restart the server.

You can move the SampleServer.jar and the SampleServer.properties file to any platform that has a JVM installed so that these files can be accessed via a network connection.

Starting the Sample Server

You start the Sample Server from a command prompt.

To start the Sample Server

1. Change the working directory to:

Integration Server_directory \packages\WmSampleAdapter\backendResource

where *Integration Server_directory* is the directory in which you installed Integration Server.

2. Execute the script **startSampleServer**.

Banking Functions

The Sample Server provides the following banking functions. You use the adapter service template to create adapter service nodes that use these functions:

Function	Description
Deposit	Creates multiple cash and check deposits. When the check number is equal to or less than 0, the deposit is considered to be a cash deposit.
Bounce Check	Bounces (disapproves) a check deposit.
Clear Check	Clears (approves) a check deposit.
Get Balance	Retrieves the current balance of the specified account.
Get History	Retrieves the service history of the specified account.
Transfer	Transfers funds between two accounts. The user ID and Personal Identification Number (PIN) are authenticated against both accounts.
Withdraw	Withdraws funds from the specified account. A negative amount cannot be withdrawn.

Banking Event Queries

Banking event queries retrieve information about specific activities that occur in the Sample Server as a result of banking function requests. Each occurrence of a banking event will be returned only once. The Sample Adapter uses polling notifications to retrieve these events and to generate notification documents.

The Sample Server supports the following banking event queries:

- CheckDepositStatusChange

  This event query requests a list of checks that have cleared or bounced since the last time the query was made. These events occur as a result of the Clear Check or Bounce Check functions.

- UnderBalance

  This event query requests a list of accounts that have been overdrawn since the last time the query was made. These events occur as a result of Withdraw or Transfer functions that result in a negative account balance. Note that the function request must have succeeded if the function request was rejected because the debit would exceed the account's credit limit. In this case, no UnderBalance event is recorded.

Because there is no protocol support for the Sample Server client to acknowledge the polling message from the receiving side, a document will not be published with the same message twice. For example, if a document reports that an account has fallen to -10, no other document is published until the amount changes.

Banking Alerts

Banking alerts are generated when specified events occur on the Sample Server. Unlike banking event queries, banking alerts are generated and delivered in real time to a configured address. The Sample Adapter implements listeners to monitor these addresses and to retrieve the alert data. Listener notifications are then used to distribute the alert information to Integration Server services. There is no queueing mechanism for alerts, so if no listener is ready to retrieve the alert data, it will be lost.

The Sample Server supports the following types of alerts:

- CheckDepositNotification

  This alert is generated for each check that is successfully deposited with the Deposit function.

- UnderBalanceNotification

  This alert is identical to the UnderBalance banking event, but is delivered as an alert.

Prerequisites for Code Compilation

Following are the prerequisites for compiling the Sample Adapter package:

- Have a Java SDK installed on your machine. The JVM installed with the webMethods software does not include a Java compiler.
- Update your jcode script in the *Integration Server_directory* \bin directory as follows:
  - Have the JAVA_DIR defined for a full JDK directory. You will need access to a Java compiler.
  - In the very last line of the command that executes the NodeUtil class, you must add the following paths to the classpath in order to compile the code correctly:
    - *Integration Server_directory* \packages\WmART\code\jars\wmart.jar
    - *Integration Server_directory* \packages\WmSampleAdapter\code\jars\SampleClient.jar
    - *Integration Server_directory* \common\lib\glassfish\gf.javax.resource.jar
    - *Integration Server_directory* \packages\WmART\code\classes

    where *Integration Server_directory* is the directory in which you installed Integration Server.

Phase 1: Creating an Adapter Definition

An adapter definition simply defines the adapter to your Integration Server. To create the adapter definition, the adapter includes the following classes:

Class	Description
WmSampleAdapter	Represents the main class of the adapter. In the Sample Adapter package's main source code directory, the adapter definition extends the base class WmAdapter.
WmSampleAdapterConstants	Contains all the string constants used by the adapter. It includes such things as the major code, group names, parameter names, bean property names, and resource domain names.
WmSampleAdapterResourceBundle	Contains all display strings and messages used by the adapter at run time and at design time.
admin	Registers and un-registers the adapter when the adapter package starts up and shuts down.

The first three classes are located in the com.wm.adapter.wmSampleAdapter Java package. The admin class is located in wm.wmSampleAdapter Java package.

Compiling the Phase 1 Implementation

To verify that the adapter definition is properly defined, compile the Phase 1 implementation (the WmSampleAdapter1 package).

Before you begin

Before you begin make sure you disable the WmSampleAdapter package.

To disable the sample adapter package

1. On the Integration Server Administrator > Packages > Management screen, make sure that the WmSampleAdapter package is disabled. To disable the package, click **Yes** in the **Enabled** column.

   The **Enabled** column now shows **No** (disabled).

   Important:
   You must disable the WmSampleAdapter and all WmSampleAdapter*N* packages before you proceed. All of these packages have the same adapter major code and conflict with each other if they are not disabled.

2. Refresh Integration Server Administrator.

   The **Sample Adapter** is no longer listed on the **Adapters** screen in Integration Server Administrator.

Creating the WmSampleAdapter1 Package

To create the WmSampleAdapter1 package

1. Start Designer.
2. Select **File** > **New**.
3. Select **Package** from the list of elements.
4. Assign the name `WmSampleAdapter1` to the package and click **Finish**.
5. Click the package name and select **File** > **Properties**.
6. In the Package Dependencies section, click to add a row and specify values for the following fields:

Field	Value
Package	`WmART`
Version	`*.*`

   Click **OK**.

7. Click **OK**.

Compiling the WmSampleAdapter1 Package

To compile the WmSampleAdapter1 package

1. Copy all the source code from the WmSampleAdapter\code\sourcePhase1 directory to the WmSampleAdapter1\code\source directory.

   Important:
   You *must* maintain the proper subdirectory structure when copying the source code files.

2. Copy the SampleClient.jar file from the WmSampleAdapter\code\jars directory to the WmSampleAdapter1\code\jars directory.

   If the WmSampleAdapter1\code\jars directory does not exist, you must create it.

3. Copy all files in WmAdapterSample\pub *except* index.html to WmSampleAdapter1\pub.

4. Make sure that you have updated your jcode script as described in "Prerequisites for Code Compilation" on page 241.

5. Execute the following commands from a command prompt:

   a. `jcode makeall WmSampleAdapter1`

   b. `jcode fragall WmSampleAdapter1`

6. Go to Integration Server Administrator > Package > Management and reload the WmSampleAdapter1 package.

Testing the WmSampleAdapter1 Package

To test the WmSampleAdapter1 package

1. In Designer, select **File** > **Refresh**.

2. Select the WmSampleAdapter1 package from the Package Navigator.

3. Go to **Properties** > **Startup/Shutdown Services** and assign the following services:

   - Assign wm.wmSampleAdapter.admin:StartUp as the startup service.
   - Assign wm.wmSampleAdapter.admin:ShutDown as the shutdown service.

4. Go to Integration Server Administrator > Package > Management and reload the WmSampleAdapter1 package.

5. Refresh Integration Server Administrator.

   The **Sample Adapter** is listed on the **Adapters** screen in Integration Server Administrator.

Disabling or Deleting Your Copy of the Phase 1 Implementation

After you compile and test the Phase 1 code, you must disable or delete the WmSampleAdapter1 package before you compile and test Phase 2. Not disabling or deleting the package results in a conflict of major codes when you compile and test Phase 2.

Phase 2: Adding a Connection

In this phase, the sample provides a connection template that connects clients to the Sample Server. This section describes how to:

- Implement the connection template. See "Implementing the Connection Template" on page 245.
- Compile the Phase 2 implementation (the WmSampleAdapter2 package). See "Compiling the Phase 2 Implementation" on page 246.
- Use the template to configure a connection node, and then test the connection node. See "Configuring the Connection Node" on page 247.

Implementing the Connection Template

To define the connection template, the adapter includes the following classes in the com.wm.adapter.wmSampleAdapter.connection package:

- WmSampleConnection, which extends the com.wm.adk.connectionWmManagedConnection class
- WmSampleConnectionFactory, which extends the com.wm.adk.connectionWmManagedConnectionFactory class

The bean properties declared in the WmSampleConnectionFactory class are:

Property	Description
sampleServerHostName	The IP host name for the computer where the Sample Server is running.
sampleServerPortNumber	The TCP/IP port number that Sample Server is accepting client connection. The default value is `4444`.
timeout	The number of milliseconds that the Integration Server will wait to obtain a connection with the Sample Server before it times out and returns an error. The default value is `20000`.
transactionType	A flag indicating whether the connection will request local transaction control (true) or an auto commit mode (false).

The WmSampleLocalTransaction class is implemented to support the configurable local transaction control for the WmSampleConnection.

Revised Code From Phase 1

In Phase 2, the existing classes from Phase 1 are modified to include the following revisions. The classes contain comments that detail the changes.

Class	Revision
WmSampleAdapter	Added a reference to WmSampleConnectionFactory in the fillAdapterTypeInfo method.

Class	Revision
WmSampleAdapterConstants	Added string constants for the connection property names.
WmSampleAdapterResourceBundle	Added entries for the connection property configuration.

Compiling the Phase 2 Implementation

To compile the Phase 2 implementation (the WmSampleAdapter2 package), use the procedure in "Compiling the Phase 1 Implementation" on page 242, substituting each occurrence of WmSampleAdapter1 with WmSampleAdapter2, and sourcePhase1 with sourcePhase2.

Important:
You must disable the WmSampleAdapter and all WmSampleAdapter*N* packages before you proceed. All of these packages have the same adapter major code and conflict with each other if they are not disabled.

Creating the TestSampleAdapter2 Package

To create the TestSampleAdapter2 package

1. Start Designer.
2. Select **File** > **New**.
3. Select **Package** from the list of elements.
4. Assign the name `TestSampleAdapter2` to the package and click **Finish**.
5. Click the package name and select **File** > **Properties**.
6. In the Package Dependencies section, click to add a row and specify values for the following fields:

Field	Value
Package	`WmART`
Version	`*.*`

   Click **OK**.
7. Click to add an additional row and specify values for the following fields:

Field	Value
Package	`WmSampleAdapter2`
Version	`*.*`

Click **OK**.

8. Click **OK**.

Configuring the Connection Node

Perform the following procedure to configure the connection node.

To configure the connection node

1. On the Integration Server Administrator > Adapters screen, click **Sample Adapter**.

   The Sample Adapter management screen opens.

2. Select **Connections**.

3. Click **Configure New Connection**.

4. On the Connection Types screen, click **Sample Server Connection**.

5. On the Configure Connection Type screen, provide values for the connection's parameters.

   a. Complete the Configure Connection Type > Sample Adapter section as follows:

Field	Description
Package	Select the namespace node package **TestSampleAdapter2**. This is the package in which you will create the connection.
Folder Name	Type the folder name `connections`. This is the folder in which you will create the connection.
Connection Name	Type the connection name `sampleConnection`.

   b. Complete the Connection Properties section as follows:

Field	Description
Sample Server Host Name	Type `localhost`.
Sample Server Port Number	Accept the default value `4444`.

Field	Description
Local Transaction Control?	Select **false**.
Sample Connector Timeout	Accept the default value `20000`.

c. Complete the Connection Management Properties section as follows:

Field	Description
Enable Connection Pooling	Accept the default value `true`, which enables the connection to use connection pooling.
Minimum Pool Size	Accept the default value `1`, which specifies the number of connections to create when the connection is enabled.
Maximum Pool Size	Accept the default value `10`, which specifies the maximum number of connections that can exist at one time in the connection pool.
Pool Increment Size	Accept the default value `1`, which specifies the number of connections by which the pool will be incremented if connections are needed, up to the maximum pool size.
Block Timeout	Specify the value `20000`, which specifies the number of milliseconds that the Integration Server will wait to obtain a connection with the Sample Server before it times out and returns an error.
Expire Timeout	Specify the value `20000`, which specifies the number of milliseconds that inactive connections can remain in the pool before they are closed and removed from the pool.
Startup Retry Count	Accept the default value `0`, which specifies the number of times that the system should attempt to initialize the connection pool at startup if the initial attempt fails, before issuing an AdapterConnectionException. The default value `0` means that the system makes a single attempt.
Startup Backoff Timeout	Accept the default value `10`, which specifies the number of seconds to wait between each attempt to initialize the connection pool. This field is irrelevant if the value of **Startup Retry Count** is set to `0`.

6. Click **Test Connection**.

   The connection is tested based on the settings provided.

7. Click **Save Connection**.

The connection name is now listed on the adapter's Connections screen and in the Service Browser of Designer.

Enabling the Connection Node

To enable the connection node

1. Start the Sample Server from a command prompt, as described in "Starting the Sample Server" on page 239.

2. To enable the connection node, on the adapter's Connections screen, click **No** in the **Enabled** column, the value changes to **Yes** (enabled).

   The server initializes a connection pool based on the provided settings. Enabling and disabling a connection node for SampleAdapter should produce entries similar to these in the server log:

```
2003-08-18 08:21:36 EDT [ART.0118.5505V1]
Adapter Runtime (Connection): Starting
connection connections:sampleConnection.
2003-08-18 08:21:36 EDT [ART.0118.5517V1]
Adapter Runtime (Connection): Creating
connection manager properties:
>>>BasicData:poolable=true,minimumPoolSize=1,maximumPoolSize=10,poolIncre
mentSize=1,blockingTimeout=20000,expireTimeout=20000,selectionSize=1<<<.
2003-08-18 08:21:36 EDT [ADA.0502.0101D] Initializing
Sample Connection
2003-08-18 08:21:36 EDT [SCC.0126.0001E] SCC
ConnectionManager Pool Started
2003-08-18 08:24:10 EDT [ART.0118.5510V1]
Adapter Runtime (Connection): Stopping
connection connections:sampleConnection.
```

   Note:
   If a connection node is enabled when the server shuts down, it will be enabled at server startup.

Disabling or Deleting Your Copy of the Phase 2 Implementation

After you compile and test the Phase 2 code, you must disable or delete the WmSampleAdapter2 and TestSampleAdapter2 packages before you compile and test Phase 3. Not disabling or deleting the packages results in a conflict of major codes when you compile and test Phase 3.

Phase 3: Adding Adapter Services

In this phase, the adapter includes the definition for an adapter service template that you can configure to execute the banking functions against the Sample Server. This section describes how to:

- Implement the adapter service template. See "Implementing the Adapter Service Template" on page 250.
- Compile the Phase 3 implementation (the WmSampleAdapter3 package). See "Compiling the Phase 3 Implementation" on page 252.

Implementing the Adapter Service Template

To define an adapter service template, you extend the class com.wm.adk.cci.interaction.WmAdapterService. For the Sample Adapter, the subclass FunctionInvocation was added in the com.wm.adapter.wmSampleAdapter.service Java package. When you configure this service template, it will be able to execute the functions against the Sample Server.

The adapter includes a utility class, DocumentHelp, in the com.wm.adapter.wmSampleAdapter.util Java package. This utility facilitates the data structure conversion from a Sample Server document and the Integration Server IData, and helps to decipher the adapter service and notification signature metadata received from the Sample Server repository.

The bean properties declared in the FunctionInvocation class are as follows:

Property	Description
functionName	The service function name. For a list of the functions, see "Banking Functions" on page 240.
inputParameterNames	The fully qualified input parameter names, including all the record structures and array indicators.
inputFieldNames	The fully qualified suggested input parameter signature names, including all the record structures and array indicators. **Note:** If you set the Boolean flag to true in the createFieldMap method, the adapter user will have the option to overwrite the suggested names. In this implementation, the user cannot change the names.
inputFieldTypes	The input parameter data types, including all the array indicators.
outputParameterNames	The fully qualified output parameter names, including all the record structures and array indicators.
outputFieldNames	The fully qualified suggested output parameter signature names, including all the record structures and array indicators. You cannot change these names.
outputFieldTypes	The output parameter data types, including all the array indicators.

The resource domains declared in the FunctionInvocation class are as follows:

Resource Domain Name	Description
functionNames	Looks up the list of service function names. For a list of the functions, see "Banking Functions" on page 240.
inputParameterNames	Looks up the fully qualified input parameter names, including all the record structures and array indicators.
inputFieldTypes	Looks up the input parameter data types, including all the array indicators.
outputParameterNames	Looks up the fully qualified output parameter names, including all the record structures and array indicators.
outputFieldTypes	Looks up the output parameter data types, including all the array indicators.

Implementing the execute Method in the FunctionInvocation Class

When the flow service invokes an adapter service node, the service calls the WmAdapterService.execute method. This method receives a WmManagedConnection object and a WmRecord object, and it is expected to return a WmRecord object.

The execute method uses the Sample Server client API to create a request document with the input parameters and sends it to the Sample Server. The request can receive one of three possible responses:

- Success with output: The service succeeds, and receives an acknowledgment document and output. For example, a getBalance service returns an account balance.
- Success with no output: The service succeeds, and receives an acknowledgment document, but there is no output. For example, a Deposit service simply deposits an amount, but returns no output.
- Failure: The service fails and receives a negative acknowledgment document; an AdapterException is thrown with the appropriate error message.

Revised Code From Phases 1 and 2

In Phase 3, the existing classes from Phase 1 and Phase 2 are modified to include the following revisions. The classes contain comments that detail the changes.

Class	Revision
WmSampleAdapterConstants	Added string constants for the service property names.
WmSampleAdapterResourceBundle	Added entries for the service property configuration.

Class	Revision
WmSampleConnectionFactory	Added a reference to FunctionInvocation in the fillResourceAdapterMetadataInfo method.
WmSampleConnection	In the registerResourceDomain method, the sample code now registers all resource domains declared by the FunctionInvocation service template. In the adapterResourceDomainLookup method, the sample code now includes resource domain lookup code to request the service metadata from the Sample Server repository.

Compiling the Phase 3 Implementation

To compile the Phase 3 implementation (the WmSampleAdapter3 package), use the procedure in "Compiling the Phase 1 Implementation" on page 242, substituting each occurrence of WmSampleAdapter1 with WmSampleAdapter3, and sourcePhase1 with sourcePhase3.

Important:
You must disable the WmSampleAdapter and all WmSampleAdapter*N* packages before you proceed. All of these packages have the same adapter major code and conflict with each other if they are not disabled.

You can use the FunctionInvocation adapter service template to create adapter service nodes that use any of the banking functions (Deposit, GetBalance, Withdraw, and so on). At a minimum, configure nodes for Deposit and either Withdraw or Transfer. You will need to execute these nodes later, when you test the notifications.

The following procedures describe how to configure and test the Deposit adapter service node. To configure and test other nodes, repeat these procedures, substituting the appropriate function name in each node.

Creating the TestSampleAdapter3 Package

To create the TestSampleAdapter3 package

1. Start Designer.
2. Select **File** > **New**.
3. Select **Package** from the list of elements.
4. Assign the name `TestSampleAdapter3` to the package and click **Finish**.
5. Click the package name and select **File** > **Properties**.

6. In the Package Dependencies section, click to add a row and specify values for the following fields:

Field	Value
Package	`WmART`
Version	`*.*`

Click **OK**.

7. Click to add an additional row and specify values for the following fields:

Field	Value
Package	`WmSampleAdapter3`
Version	`*.*`

Click **OK**.

8. Click **OK**.

Configuring the Deposit Adapter Service Node

Perform the following procedure to configure the Deposit adapter service node.

To configure the Deposit adapter service node

1. On the Integration Server Administrator > Adapters screen, click **Sample Adapter**.

   The Sample Adapter management screen opens.

2. Select **Connections**.

3. Configure the sampleConnection connection node in the TestSampleAdapter3 package, and enable the connection, as described in "Configuring the Connection Node" on page 247.

4. Go to Integration Server Administrator > Package > Management and reload the WmSampleAdapter3 package.

5. In Designer, select **File** > **Refresh**.

6. In the Package Navigator, navigate to the **TestSampleAdapter3** package and create a folder named `services`.

7. Select **File** > **New**.

8. Select **Adapter Service** from the list of elements.

9. On the Create a New Adapter Service screen, type `deposit` in the **Element name** field and click **Next**.

10. On the Select Adapter Type screen, select **Sample Adapter** and click **Next**.

11. On the Select an Adapter Connection Alias screen, select **connections:sampleConnection** and click **Next**.

12. On the Select a Template screen, select **Function Invocation** and click **Next**.

13. Assign the name to the service and click **Finish**.

    The adapter creates the adapter service, and the Adapter Service Editor appears.

14. Select **File** > **Save**.

15. In the Adapter Service Editor's **Function Invocation** tab, select **Deposit** from the **Function Name** field.

16. Accept all default values on the **Settings** tab. For more information, see the *webMethods Service Development Help* for your release.

17. Select **File** > **Save**.

Note:
To create additional adapter service nodes, repeat this procedure, selecting the appropriate function names, such as Withdraw or Transfer.

Testing the Deposit Adapter Service Node

Perform the following procedure to test the Deposit adapter service node.

To test the Deposit adapter service node

1. Start Designer.

2. Click the **deposit** service in the **services** folder to add a record to the Sample Server.

3. On the **Test** menu, select **Run**.

4. Enter data in the fields of the pop-up menu that appears as follows, and click **OK**.

Field	Value
User ID	uid1
PIN	pin1

5. Enter data in the fields of the Add Row pop-up menu that appears as follows, and click **OK**.

Field	Value
Deposit To Account	1
Deposit Amount	10
Check Number	`0` **Note:** Any value less than or equal to 0 specifies a cash deposit. Specify 0 for this test because a check deposit amount is not added to the balance until the check deposit is cleared (approved).

6. View your input data in the **Results** tab.

Disabling or Deleting Your Copy of the Phase 3 Implementation

After you compile and test the Phase 3 code, you must disable or delete the WmSampleAdapter3 and TestSampleAdapter3 packages before you compile and test Phase 4. Not disabling or deleting the packages results in a conflict of major codes when you compile and test Phase 4.

Phase 4: Adding Polling Notifications

In this phase, the adapter includes the definition for a polling notification template that you use to configure polling notification nodes. These nodes will poll the Sample Server to determine whether checks have cleared or bounced, or whether accounts have negative balances. This section describes how to:

- Implement the polling notification template. See "Implementing the Polling Notification Template" on page 256.
- Compile the Phase 4 implementation (the WmSampleAdapter4 package). See "Compiling the Phase 4 Implementation" on page 258.
- Use the template to configure two polling notification nodes (one for check clearing/bouncing, the other for negative balances).

  Each node generates a document that will be used to contain the affected portion of the Sample Server data, and to inform the Integration Server of the changes. See "Configuring the underBalancePolling Notification Node" on page 259.

- Create an Integration Server trigger and a flow service for each polling notification node.

  The notifications will publish their result documents to the triggers. Upon receiving a document generated by the polling notification, the trigger causes the Integration Server to invoke a flow service registered with the trigger to process the document's data. In the Sample Adapter, the flow service invokes the pub.flow:savePipelineToFile service. This service simply saves the polling notification event (the contents of the pipeline) to a file. Typically, this service is used as a debugging tool. It is provided here simply to demonstrate the use of the notification. In a real adapter, you would typically perform some kind of action with the notification data. See "Creating the Trigger for the underBalancePolling Notification Node" on page 261 and "Creating the Flow Service for the underBalancePolling Notification Node" on page 261.

- Schedule and enable the polling notification nodes. See "Scheduling and Enabling the underBalancePolling Notification Node" on page 262.

- Test the polling notification nodes. See "Testing the underBalancePolling Notification Node" on page 262.

Implementing the Polling Notification Template

To define a polling notification template, you extend the class com.wm.adk.notification.WmPollingNotification. The Sample Adapter includes the subclass as MessagePolling in the com.wm.adapter.wmSampleAdapter.notification Java package. You will use this template to configure two nodes to monitor the following types of events:

- CheckDepositStatusChange

  The Sample Server records this event when a Clear Check or Bounce Check service clears (approves) or bounces (disapproves) a deposited check.

- UnderBalance

  The Sample Server records this event when a Withdraw or Transfer service causes a negative account balance. The document will not be published if the account's negative balance is greater than the available credit limit amount. For example, if the credit limit amount is 1000, and the account balance falls to -1001, the document is not published; an error is issued.

The bean properties declared in the MessagePolling class are as follows:

Property	Description
pollingName	The event to poll in the Sample Server.
inputParameterNames	The fully qualified input parameter names, including all the record structures and array indicators.
inputFieldValues	Unlike adapter services, polling notifications accept no run-time input data, other than the configured property values specified here.
inputFieldTypes	The input parameter data types, including all the array indicators.

Property	Description
outputParameterNames	The fully qualified output parameter names, including all the record structures and array indicators.
outputFieldNames	The fully qualified suggested output parameter signature names, including all the record structures and array indicators. You cannot change these names.
outputFieldTypes	The output parameter data types, including all the array indicators.

The resource domains declared in the MessagePolling class are as follows:

Resource Domain	Description
pollingNames	Looks up the list of Sample Server polling notification event names described in "Banking Event Queries" on page 240.
inputParameterNames	Looks up the fully qualified input parameter names, including all the record structures and array indicators.
inputFieldTypes	Looks up the input parameter data types, including all the array indicators.
outputParameterNames	Looks up the fully qualified output parameter names, including all the record structures and array indicators.
outputFieldTypes	Looks up the output parameter data types, including all the array indicators.

Implementing the runNotification Method

The template includes the runNotification method, which is called by the Integration Server based on the polling notification node's schedule. This method has no arguments or return values; it merely publishes documents.

This method constructs an event query document with the query criteria input, and sends it to the Sample Server. It then waits for the reply document from the Sample Server. The query can produce one of three possible responses:

- Success with output: The notification succeeds, and receives a notification document.
- Success with no output: The notification succeeds, but no event has occurred. The notification receives an acknowledgment document.
- Failure: The notification fails and receives a negative acknowledgment document; an AdapterException is thrown with the appropriate error message.

Revised Code From Phases 1, 2, and 3

In Phase 4, the existing classes from Phases 1, 2, and 3 are modified to include the following revisions. The classes contain comments that detail the changes.

Class	Revision
WmSampleAdapterConstants	Added string constants for the polling property names.
WmSampleAdapterResourceBundle	Added entries for the polling property configuration.
WmSampleConnection	In the registerResourceDomain method, the adapter code now registers all the resource domains declared by the MessagePolling class. In the adapterResourceDomainLookup method, the adapter code now includes resource domain lookup code to request the polling metadata from Sample Server repository.

Compiling the Phase 4 Implementation

To compile the Phase 4 implementation (the WmSampleAdapter4 package), use the procedure in "Compiling the Phase 1 Implementation" on page 242, substituting each occurrence of WmSampleAdapter1 with WmSampleAdapter4, and sourcePhase1 with sourcePhase4.

Important:
You must disable the WmSampleAdapter and all WmSampleAdapter*N* packages before you proceed. All of these packages have the same adapter major code and conflict with each other if they are not disabled.

Configuring and Testing the Polling Notification Nodes

You can create polling notification nodes to send banking event queries to the Sample Server. To create these nodes, you use the MessagePolling polling notification template. The two banking event query types are:

- UnderBalance
- CheckDepositStatusChange

The following procedures describe how to configure and test a node for UnderBalance. To configure and test a node for CheckDepositStatusChange, repeat these procedures, substituting the polling name CheckDepositStatusChange for UnderBalance.

This section contains the following procedures:

- "Configuring the underBalancePolling Notification Node" on page 259.
- "Creating the Trigger for the underBalancePolling Notification Node" on page 261.
- "Creating the Flow Service for the underBalancePolling Notification Node" on page 261.

- "Scheduling and Enabling the underBalancePolling Notification Node" on page 262.
- "Testing the underBalancePolling Notification Node" on page 262.

Creating the TestSampleAdapter4 Package

To create the TestSampleAdapter4 package

1. Start Designer.
2. Select **File** > **New**.
3. Select **Package** from the list of elements.
4. Assign the name `TestSampleAdapter4` to the package and click **Finish**.
5. Click the package name and select **File** > **Properties**.
6. In the Package Dependencies section, click to add a row and specify values for the following fields:

Field	Value
Package	`WmART`
Version	`*.*`

   Click **OK**.

7. Click to add an additional row and specify values for the following fields:

Field	Value
Package	`WmSampleAdapter4`
Version	`*.*`

   Click **OK**.

8. Click **OK**.

Configuring the underBalancePolling Notification Node

Perform the following procedure to configure the underBalancePolling notification node.

To configure the underBalancePolling notification node

1. On the Integration Server Administrator > Adapters screen, click **Sample Adapter**.

   The Sample Adapter management screen opens.

2. Select **Connections**.

3. Configure the sampleConnection connection node in the TestSampleAdapter4 package, and enable the connection, as described in "Configuring the Connection Node" on page 247.

4. In Designer, select **File** > **Refresh**.

5. In the Package Navigator, navigate to the **TestSampleAdapter4** package and create a folder named `pollingNotifications`.

6. Select **File** > **New**.

7. Select **Adapter Notification** from the list of elements.

8. On the Create a New Adapter Notification screen, type `underBalancePolling` in the **Element name** field and click **Next**.

9. On the Select Adapter Type screen, select **Sample Adapter** and click **Next**.

10. On the Select a Template screen, select **Message Polling** and click **Next**.

11. On the Select an Adapter Connection Alias screen, select **connections:sampleConnection** and click **Next**.

12. Click **Finish**.

    Designer creates the notification and the publish document named **underBalancePollingPublishDocument**.

13. In the Adapter Notification Editor's **Message Polling** tab, select the polling event **UnderBalance** from the **Polling Name** field.

14. Specify values for the following fields in the **Input Field Value** column for the corresponding Input Parameters, and click **Save**.

Field	Value
User ID	`suid` (the "super user" ID)
PIN	spin (the "super user" PIN)
Account Number	`0` (enables polling against all accounts)

15. Select **File** > **Save**.

Creating the Flow Service for the underBalancePolling Notification Node

Perform the following procedure to create the flow service for the underBalancePolling notification node.

To create the flow service for the underBalancePolling notification node

1. Start Designer.
2. In the Package Navigator, navigate to the **TestSampleAdapter4** package and select the pollingNotifications folder.
3. Select **File** > **New**.
4. Select **Flow Service** from the list of elements.
5. On the Create a New Adapter Service screen, type `underBalancePollingService` in the **Element name** field and click **Next**.
6. On the Select the Source Type screen, select **Empty Flow** and click **Finish**.
7. Click ⇨ to insert a flow step.
8. Navigate to the **pub.flow:savePipelineToFile** service in the **WmPublic** package and click **OK**.

   Note:
   The savePipelineToFile service saves the polling notification event (the contents of the pipeline) to the file that you specify in the **fileName** parameter.
9. Click the **Pipeline** tab.
10. Open the **fileName** parameter in the pipeline and set its value to `underBalancePolling.log`.

    Click **OK**.

Creating the Trigger for the underBalancePolling Notification Node

Perform the following procedure to create the trigger for the underBalancePolling notification node.

To create the trigger for the underBalancePolling notification node

1. Start Designer.

2. In the Package Navigator, navigate to the **TestSampleAdapter4** package and select the pollingNotifications folder.

3. Select **File** > **New**.

4. Select **Broker/Local Trigger** from the list of elements.

5. On the Create a New Broker/Local Trigger screen, type `underBalancePollingTrigger` in the **Element name** field and click **Finish**.

6. In the trigger editor, in the Conditions section, accept the default **Condition1**.

7. In the Condition detail section, in the **Service** field, select or type the flow service name `pollingNotifications:underBalancePollingService`.

8. Click  to insert document types. Select **underBalancePollingPublishDocument** and click **OK**.

9. Click  to save your trigger.

Scheduling and Enabling the underBalancePolling Notification Node

Perform the following procedure to schedule and enable the underBalancePolling notification node.

To schedule and enable the underBalancePolling notification node

1. On the Integration Server Administrator > Adapters screen, click **Sample Adapter**.

   The Sample Adapter management screen opens.

2. Select **Polling Notifications**.

3. Click **Edit Schedule**.

4. Set the **Interval** to `10` and click the **Save Schedule** button.

5. Enable the node by selecting **Enabled** in the **State** column.

Testing the underBalancePolling Notification Node

Perform the following procedure to test the underBalancePolling notification node.

To test the underBalancePolling notification node

1. Invoke a Withdraw service or a Transfer service against an account to cause a negative account balance. You create a Withdraw service or a Transfer service similarly to the Deposit service described in "Compiling the Phase 3 Implementation" on page 252.

   Note:
   No banking event is generated.

2. Search the underBalancePolling.log file in the *Integration Server_directory* \pipeline directory for the notification message. This is the file you specified as the value of the fileName parameter in "Creating the Flow Service for the underBalancePolling Notification Node" on page 261.

   Note:
   The adapter code uses the savePipelineToFile service to save the polling notification event to a file. A new polling notification event of the same type will overwrite the contents of the corresponding log file.

Disabling or Deleting Your Copy of the Phase 4 Implementation

After you compile and test the Phase 4 code, you must disable or delete the WmSampleAdapter4 and TestSampleAdapter4 packages before you compile and test Phase 5. Not disabling or deleting the packages results in a conflict of major codes when you compile and test Phase 5.

Phase 5: Adding Listener Notifications

This phase represents the adapter with all its functionality. In this phase, the adapter now includes the definition for a listener notification template that you can use to configure listener notification nodes. These nodes will receive alerts immediately from the Sample Server when checks have been deposited and when accounts have negative balances. This section describes how to:

- Implement the listener.

  A listener object is connected to the adapter resource, waiting for the system to deliver notifications. See "Implementing the Listener" on page 264.

- Implement the asynchronous listener notification template.

  An asynchronous listener notification publishes a document to a webMethods Broker queue, using the doNotify method. You will create a trigger that receives the document and executes an Integration Server flow service to process the document's data. See "Implementing the Asynchronous Listener Notification Template" on page 264.

- Configure the listener. See "Configuring the Listener Node" on page 267.

- Use the template to configure two asynchronous listener notification nodes (one for check deposits, the other for negative balances). See "Configuring the checkDepositListener Notification Node" on page 268.

- Create an Integration Server trigger and a flow service for each listener notification node.

The notifications publish their result documents to the triggers. Upon receiving a document, the trigger causes the Integration Server to invoke a flow service registered with the trigger to process the data contained in the document. In the Sample Adapter, the flow service invokes the pub.flow:savePipelineToFile service. This service simply saves the notification event to a file. Typically, this service is used as a debugging tool. It is provided here simply to demonstrate the use of the notification. In a real adapter, you would typically perform some kind of action with the notification data. See "Creating the Trigger for the checkDepositListener Notification Node" on page 270 and "Creating the Flow Service for the checkDepositListener Notification Node" on page 269.

- Test the listener notification nodes. See "Testing the checkDepositListener Notification Node" on page 271.

Note:
There is no need to compile the Phase 5 implementation. The WmSampleAdapter package represents the Phase 5 (final) implementation package as delivered.

Implementing the Listener

The listener is a Sample Server client that has a connection in the passive listening mode. To define a listener, you extend the class com.wm.adk.notification.WmConnectedListener. The Sample Adapter includes the subclass as WmSampleListener in the com.wm.adapter.wmSampleAdapter.connection Java package. When this listener is configured, it will be able to connect and monitor the Sample Server for the occurrence of specified events.

Normally, a listener does not create its native connection to the back-end system. Instead, it calls the method retrieveConnection to retrieve a connection object that you specify when you configure the listener. In the waitForData method, the connection object will wait and return the notification event. If the listener times out, it returns null. The time out of the connection while in the blocked reading mode is an important feature of the connection object. It gives the adapter the opportunity to shut down the listener if it is disabled.

Implementing the Asynchronous Listener Notification Template

To define an asynchronous listener notification template, you extend the class com.wm.adk.notification.WmAsyncListenerNotification. The Sample Adapter includes the subclass as AsyncListening in the com.wm.adapter.wmSampleAdapter.notification Java package. You will use this template to configure a node for each of the following alert types:

- CheckDepositNotification

  The system publishes this notification document when the Deposit service successfully deposits a check. You can then invoke the Clear Check service or the Bounce Check service to approve or disapprove the check.

- UnderBalanceNotification

  This notification operates under the same criteria as the UnderBalance polling notification (see "Implementing the Polling Notification Template" on page 256).

The following WmAsyncListenerNotification methods are implemented:

- The supports method, which compares the notification event name in the notification with the configured notification name to determine whether it should claim the event or not. If it is appropriate, it will return true. Otherwise, it will return false.
- The runNotification method, which processes the notification event and publishes the event to the webMethods Broker queue.

The WmSampleListener class does not declare its own bean properties. The properties declared in the AsyncListening class are as follows:

Property	Description
eventName	The event notification type name.
outputParameterNames	The fully qualified output parameter names, including all the record structures and array indicators.
outputFieldNames	The fully qualified suggested output parameter signature names, including all the record structures and array indicators. **Note:** If you set the Boolean flag to true in the createFieldMap method, the adapter user will have the option to overwrite the suggested names. In this implementation, the user cannot change the names.
outputFieldTypes	The output parameter data types, including all the array indicators.

The resource domains declared in the AsyncListening class are as follows:

Resource Domain	Description
notificationNames	Looks up the list of the Sample Server listener notification event names described in "Banking Alerts" on page 241.
outputParameterNames	Looks up the fully qualified output parameter names, including all the record structures and array indicators.
outFieldTypes	Looks up the output parameter data types, including all the array indicators.

Revised Code From Phases 1, 2, 3, and 4

In Phase 5, the existing classes from Phases 1 through 4 are modified to include the following revisions. The classes contain comments that detail the changes.

Class	Revision
WmSampleAdapterConstants	Added string constants for the polling property names.

Class	Revision
WmSampleAdapterResourceBundle	Added entries for the polling property configuration.
WmSampleConnection	In the registerResourceDomain method, the sample code now registers all the resource domains declared by the MessagePolling template. In the adapterResourceDomainLookup method, the sample code now includes resource domain lookup code to request the polling metadata from Sample Server repository.

Configuring and Testing the Listener and the Listener Notification Nodes

You can use the AsyncListening listener notification template to create listener notification nodes that monitor the alert types generated by the Sample Server. The two alert types are:

- CheckDepositNotification
- UnderBalanceNotification

The following procedures describe how to configure and test a node for CheckDepositNotification. To configure and test a node for UnderBalanceNotification, repeat these procedures, substituting the polling name UnderBalanceNotification for CheckDepositNotification.

This section describes the following tasks:

- "Creating the TestSampleAdapter Package" on page 266
- "Configuring the Listener Node" on page 267.
- "Configuring the checkDepositListener Notification Node" on page 268.
- "Creating the Trigger for the checkDepositListener Notification Node" on page 270.
- "Creating the Flow Service for the checkDepositListener Notification Node" on page 269.
- "Testing the checkDepositListener Notification Node" on page 271.

Creating the TestSampleAdapter Package

To create the TestSampleAdapter package

1. Start Designer.
2. Select **File** > **New**.

3. Select **Package** from the list of elements.

4. Assign the name `TestSampleAdapter` to the package and click **Finish**.

5. Click the package name and select **File** > **Properties**.

6. In the Package Dependencies section, click to add a row and specify values for the following fields:

Field	Value
Package	`WmART`
Version	`*.*`

   Click **OK**.

7. Click to add an additional row and specify values for the following fields:

Field	Value
Package	`WmSampleAdapter`
Version	`*.*`

   Click **OK**.

8. Click **OK**.

Configuring the Listener Node

Perform the following procedure to configure the listener node.

To configure the listener node

1. On the Integration Server Administrator > Adapters screen, click **Sample Adapter**.

   The Sample Adapter management screen opens.

2. Select **Connections**.

3. Configure a new connection node named `listenerConnection` in the TestSampleAdapter package, and enable the connection, as described in "Configuring the Connection Node" on page 247.

4. On the Sample Adapter management screen, click **Listeners**.

5. Click **Configure New Listener**.

6. On the Listener Types screen, click **Sample Server Listener**.

7. Complete the Configure Listener Type > Sample Adapter section as follows:

Parameter	Description
Package	Select the namespace node package **TestSampleAdapter**. This is the package in which you will create the listener.
Folder Name	Type the folder name `listeners`. This is the folder in which you will create the listener.
Listener Name	Type the listener name `sampleListener`.
Connection	Select the connection node **connections:listenerConnection**.
Retry Limit	Accept the default value `5`, which specifies the number of times that the system should attempt to start the listener if the initial attempt fails. Specifically, it specifies how many times to retry the listenerStartup method before issuing an AdapterConnectionException. The value `0` means that the system makes a single attempt.
Retry Backoff Timeout	Accept the default value `10`, which specifies the number of seconds the system should wait between each attempt to start the listener.

8. Click **Save Listener**.

Note:
Enabling the listener before you configure and enable its corresponding listener notification node produces a warning.

Configuring the checkDepositListener Notification Node

Perform the following procedure to configure the checkDepositListener notification node.

To configure the checkDepositListener notification node

1. Start Designer.

2. In the Package Navigator, navigate to the **TestSampleAdapter** package and create a folder named `listenerNotification`.

3. Select **File** > **New**.

4. Select **Adapter Notification** from the list of elements.

5. On the Create a New Adapter Notification screen, type `checkDepositListener` in the **Element name** field and click **Next**.

6. On the Select Adapter Type screen, select **Sample Adapter** and click **Next**.

7. On the Select a Template screen, select **Asynchronous Listener Notification** and click **Next**.

8. On the Select an Adapter Notification Listener screen, select **connections:sampleListener** and click **Next**.

9. Click **Finish**.

10. On the **Listener Notification** tab in the Adapter Notification Editor, select **CheckDepositNotification** in the **Asynchronous Listening Name** field.

11. Select **File** > **Save**.

Creating the Flow Service for the checkDepositListener Notification Node

Perform the following procedure to create the flow service to log the checkDepositListener notification node.

To create the flow service to log the checkDepositListener notification node

1. Start Designer.

2. In the Package Navigator, navigate to the **TestSampleAdapter** package and select the listenerNotification folder.

3. Select **File** > **New**.

4. Select **Flow Service** from the list of elements.

5. On the Create a New Adapter Service screen, type `checkDepositListenerService` in the **Element name** field and click **Next**.

6. On the Select the Source Type screen, select **Empty Flow** and click **Finish**.

7. Click ⇨ to insert a flow step.

8. Navigate to the **pub.flow:savePipelineToFile** service in the **WmPublic** package and click **OK**.

> **Note:**
> The savePipelineToFile service saves the polling notification event (the contents of the pipeline) to the file that you specify in the **fileName** parameter.

9. Click the **Pipeline** tab.

10. Open the **fileName** parameter in the pipeline and set its value to `checkDepositListener.log`.

    Click **OK**.

Creating the Trigger for the checkDepositListener Notification Node

Perform the following procedure to create the trigger for the checkDepositListener notification node.

To create the trigger for the checkDepositListener notification node

1. Start Designer.

2. In the Package Navigator, navigate to the **TestSampleAdapter** package and select the listenerNotification folder.

3. Select **File** > **New**.

4. Select **Broker/Local Trigger** from the list of elements.

5. Assign the name `checkDepositListenerTrigger` to the trigger and click **Finish**.

6. In the trigger editor, in the Conditions section, accept the default **Condition1**.

7. In the Condition detail section, in the **Service** field, select or type the flow service name `listenerNotification:checkDepositListenerService`.

8. Click  to insert document types. Select **checkDepositListenerPublishDocument** and click **OK**.

9. Click **Save**.

Enabling the checkDepositListener Notification Node and the Listener Node

Perform the following procedure to enable the checkDepositListener notification node and the listener node.

To enable the checkDepositListener notification node and the listener node

1. On the Integration Server Administrator > Adapters screen, click **Sample Adapter**.

   The Sample Adapter management screen opens.

2. Click **Listener Notifications**.

3. Enable the **checkDepositListener** notification by clicking **No** in the **Enabled** column.

   The **Enabled** column now shows **Yes** (enabled).

4. Click **Listeners**.

5. Enable the **sampleListener** by selecting **Enabled** in the **State** column.

Testing the checkDepositListener Notification Node

Perform the following procedure to test the checkDepositListener notification node.

To test the checkDepositListener notification node

1. In Designer, invoke the Deposit adapter service that you configured in "Configuring the Deposit Adapter Service Node" on page 253 to produce a check deposit event as follows:

   a. Start Designer.

   b. Click the **deposit** service in the services folder to add a record to the Sample Server.

   c. Select **Run** > **Run**.

   d. Specify values in the fields as follows:

Field	Value
User ID	`uid1`
PIN	`pin1`

   Click **OK**.

   e. Specify values for the fields of the Add Row screen as follows.

Field	Value
Deposit To Account	`1`
Deposit Amount	`10`
Check Number	`1`

   Click **OK**.

The data you entered is displayed in the **Results** tab.

2. Search the checkDepositListener.log file in the *Integration Server_directory* \pipeline directory for the check deposit notification message. (This is the file you specified as the value of the **fileName** parameter in "Creating the Flow Service for the checkDepositListener Notification Node" on page 269.)

Note:
The adapter's code uses the savePipelineToFile service to save the notification event to a file. A new notification event of the same type will overwrite the contents of the corresponding log file.