BEA WebLogic Application Integration

A Component of BEA WebLogic Integration

Adapter Development Guide

BEA WebLogic Application Integration Release 2.0
Document Edition 2.0
July 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

BEA WebLogic Application Integration

Document Edition	Part Number	Date	Software Version
2.0	N/A	July 2001	2.0

Contents

7. Developing an Event Adapter

8. Developing a Design-Time GUI

A. Adapter Setup Worksheet

B. The DBMS Sample Adapter

C. The eMail Sample Adapter

About This Document

BEA WebLogic Application Integration Adapter Development Guide is organized as follows:

- "Introduction to the ADK" provides a brief background on the BEA WebLogic Application Integration Adapter Development Kit. It discusses service and event adapters, the design-time GUI, and what to do before you start building an adapter.
- "Concepts" discusses of some of the ADK concepts relevant to adapter development, including events and services, designtime vs. runtime, logging, and the adapter logical name.
- "Tools" describes the ADK tools provided that you can use to build adapters. These tools include the sample adapter, the GenerateAdapterTemplate utility, the Ant-based build process, XML tools, and Javadoc.
- "Creating a Custom Development Environment" shows how to use the GenerateAdapterTemplate utility to clone the sample adapter and customize a development environment for your new adapter.
- "The ADK Logging Toolkit" describes how to use the ADK logging toolkit to implement logging. It also includes a discussion of the Apache log4j specification, which is the core of the ADK logging framework.
- "Developing a Service Adapter" shows you how to build an adapter that supports services. It delineates all of the steps required to successfully create the adapter and shows relevant code samples where necessary.
- "Developing an Event Adapter" shows you how to build an adapter that supports events. It delineates all of the steps required to successfully create the adapter and shows relevant code samples where necessary.
- "Developing a Design-Time GUI" shows you how to build a graphical user interface that adapter users need to define, deploy, and test their application

views. It delineates all of the steps required to successfully create the GUI and shows relevant code samples where necessary.

- "Adapter Setup Worksheet" is a worksheet that will help you conceptualize the adapter you are building before you actually began to code. It will help you define such components as the adapter logical name and the Java package base name and help you determine the locales for which you need to localize message bundles.
- "The DBMS Sample Adapter" describes how the ADK was used to build a DBMS adapter. It also contains a simple task-driven example of how to use the DBMS adapter.
- "The eMail Sample Adapter" describes how the ADK was used to build an eMail adapter. It also contains a simple task-driven example of how to use the eMail adapter.

What You Need to Know

The *BEA WebLogic Application Integration Adapter Development Guide* is designed primarily for use by adapter developers who will use the ADK to develop service adapters, event adapters, and the design-time GUI that adapter users employ to create application views.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at http://e-docs.beasys.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the BEA WebLogic Application Integration documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the BEA WebLogic Application Integration documentation Home page, click the PDF files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at `http://www.adobe.com/`.

Related Information

The following resources are also available:

- BEA WebLogic Server documentation (`http://www.edocs.com`)
- BEA WebLogic Process Integrator documentation (`http://www.edocs.com`)
- XML Schema Specification (`http://www.w3.org/TR/xmlschema-l/`)
- The Sun Microsystems, Inc. J2EE Connector Architecture Specification (`http://java.sun.com/j2ee/connector/`)

Contact Us!

Your feedback on the BEA WebLogic Application Integration documentation is important to us. Send us e-mail at **docsupport@beasys.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the BEA WebLogic Application Integration documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Application Integration 2.0 release.

If you have any questions about this version of BEA WebLogic Application Integration, or if you have problems installing and running BEA WebLogic Application Integration, contact BEA Customer Support through BEA WebSupport at **www.beasys.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.

Convention	Item
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

Convention	Item
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

. . .	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Introduction to the ADK

Welcome to the *BEA WebLogic Application Integration Development Guide*, the "how to" guide for using the BEA WebLogic Application Integration Adapter Development Kit (ADK). This guide will show you how to develop, test, and deploy event and service adapters and the design-time user interface.

This section provides information on the following subjects:

- What is the ADK?
- What are Adapters?
  - Service Adapters
  - The Design-Time GUI
- The Design-Time GUI
- Before You Begin

Section Objectives

This section serves as an overview to using the ADK to develop event and service adapters and a design-time GUI. You will learn:

- What adapters are and how they are used.
- Prerequisites you must meet before beginning adapter or GUI development.

- Terminology associated with adapter or GUI development.

What is the ADK?

The ADK is the tool set for implementing the event and service protocol of BEA WebLogic Application Integration; that is, it is a collection of frameworks that support the development, testing, packaging, and distribution of resource adapters for the WebLogic platform. Specifically, the ADK is comprised of the following four frameworks:

- Designtime
- Runtime
- Logging
- Packaging.

The ADK provides:

- Run-time support for events and services.
- An API to integrate an adapter's user interface into the Application Integration Application View Management Console.

  The added value provided by the ADK is that adapters can become an integral part of a single graphical console application that allows business users to construct integration solutions.

What are Adapters?

Resource adapters—referred to in this document as "adapters"—are software that connect one application to another when those applications are not originally designed to communicate with each other. For example, an order entry system built by one company requires an adapter to communicate with a customer information system built by another.

By using the ADK, you can create two types of adapters:

- Service adapters, which consume messages.
- Event adapters, which generate messages.

Service Adapters

Service adapters receive an XML request document from a client and invoke a specific function in the underlying enterprise information system (EIS). They are consumers of messages and may or may not provide a response. There are two ways to invoke a service: asynchronous and synchronous. With an asynchronous service adapter, the client application issues a service request and then proceeds with its processing. The client application does not wait for the response. With a synchronous service adapter, the client waits for the response before proceeding with its processing. The Application Integration platform supports both of these service adapter invocations, relieving you from having to provide this functionality.

Service adapters perform the following four functions:

- Receive service requests from an external client.
- Transform the XML request document into the EIS specific format. The request document conforms to the request XML schema for the service. The request XML schema is based on metadata in the EIS.
- Invoke the underlying function in the EIS and wait for its response.
- Transform the response from the EIS specific data format to an XML document that conforms to the response XML schema for the service. The response XML schema is based on metadata in the EIS.

As with events, the ADK implements the aspects of these four functions that are generic across all service adapters.

To learn how to develop a service adapter, see “Developing a Service Adapter.”

Event Adapters

Event adapters are designed to propagate information from an EIS into an Application Integration server. These types of adapters can be described as publishers of information.

There are two basic types of event adapters: in-process and out-of-process. In-process event adapters execute within the same process as the EIS. Out-of-process adapters execute in a separate process. In-process and out-of-process event adapters only differ in how they accomplish the data extraction process.

On the ADK integration platform, all event adapters perform the following three functions:

- Respond to "events" deemed to be of interest to some external party that occur inside the running EIS and extract data about the event from the EIS into the adapter.
- Transform event data from the EIS specific format to an XML document that conforms to the XML schema for the event. The XML schema is based on metadata in the EIS.
- Propagate the event to an event context obtained from the application view.

The ADK implements the aspects of these three functions that are generic across all event adapters. Consequently, you can focus on the EIS specific aspects of their adapter. This concept is the same as the concept behind Enterprise Java Beans (EJB). With EJB, the container provides system-level services for EJB developers so they can focus on implementing business application logic.

To learn how to develop an event adapter, see "Developing an Event Adapter."

The Design-Time GUI

Along with event and service adapters, the ADK's design-time framework provides the tools you will use to build the web-based GUI that adapter users need to define, deploy, and test their application views. Although each adapter has EIS-specific

functionality, all adapters require a GUI for deploying application views. This framework minimizes the effort required to create and deploy these interfaces, primarily by using these two components:

- A web application component that allows you to build an HTML-based GUI by using Java Server Pages (JSP). This component is augmented by tools such as the JSP templates and tag library and the JavaScript library.
- A deployment helper component, called AbstractDesignTimeRequestHandler that provides a simple API for deploying, undeploying, copying, and editing application views on a Application Integration server.

To learn how to develop a design-time GUI, see "Developing a Design-Time GUI."

The Application View

A key component of the Design-Time GUI is the *application view*. The application view represents a business-level interface to the specific functionality in an application. An adapter represents a system-level interface to all the functionality in the application. An application view is configured for a single business purpose and contains only the services related to that business purpose. These services require only business-relevant data to be specified in the request document and return only business-relevant data in the response document. Under the covers, the application view combines this business-relevant data with stored metadata necessary for the adapter. The adapter takes both the business-relevant data and the stored metadata and executes a system-level function on the application.

In addition, the application view represents both events and services that support a business purpose. This allows the business user to interact with the application view for all communication with an application. This bidirectional communication is actually supported by two adapter components (the event adapter and service adapter). The application view abstracts this fact from the user and presents them with a unified business interface to the application.

For more information about application views, see *BEA WebLogic Application Integration Product Overview*. Refer to "Application View."

The Packaging Framework

The ADK packaging framework provides a tool set for packaging an adapter for delivery to a customer. Ideally, all adapters are installed, configured, and uninstalled the same on an Application Integration server. Moreover, all service adapters must be J2EE compliant. The packaging framework makes creating a J2EE adapter archive (`.rar`) file, web application archive (`.war`) file, and Application Integration design environment archive easy.

Before You Begin

Before you can actually begin developing an adapter, be sure the ADK is installed on your computer. See the *BEA WebLogic Application Integration Installation and Configuration Guide* and the *BEA WebLogic Application Integration Release Notes* for more information.

CHAPTER

2 Concepts

This section describes some of the more important concepts with which you should familiarize yourself before attempting to develop an adapter or design-time GUI. You will see additional discussion of all of the following concepts at some point in the adapter/GUI development procedures.

This section provides information on the following subjects:

- Runtime vs. Designtime
- Events and Services
- Logging
  - The Logging Toolkit
  - The Logging Framework
  - Internationalization and Localization
- Adapter Logical Name

Runtime vs. Designtime

Adapter activity falls within one of two conceptual entities: runtime and designtime. Runtime refers to functionality that occurs when adapters execute their processes. Designtime refers to the adapter user's implementation of an application view; in essence, designtime is the act of creating, deploying, and testing an application view.

Runtime and designtime are characterized in the ADK by the run-time and design-time frameworks. The run-time framework is comprised of the tools used when developing adapters while the design-time framework refers to the tools you will use to design web-based user interfaces. Runtime and designtime are discussed in greater detail below.

Note: In this document, the spelling of “designtime” and “runtime” will vary depending upon usage. They are alternately spelled “designtime” or “runtime” (noun) and “design-time” or “run-time” (adjective).

Run-time Framework

The runtime framework is the set of tools you will use to develop event and service adapters. To support event adapter development, the runtime framework provides a basic, extensible event generator. For service adapter development, the runtime framework provides a complete J2EE-compliant adapter.

The classes supplied by the runtime framework provide the following benefits:

- They allow you to focus on EIS specifics rather than J2EE specifics.
- They minimize the effort needed to use the ADK logging framework.
- They simplify the complexity of J2EE Connector Architecture.
- They minimize redundant code across adapters.

In addition, the runtime framework provides abstract base classes to help you implement an event generator to leverage the event support provided by the ADK environment.

A key component of the run-time framework is the run-time engine, which hosts the adapter component responsible for handling service invocations and manages:

- physical connections to the EIS
- login authentication
- transaction management

all in compliance with the J2EE Connector Architecture standard.

Design-Time Framework

The design-time framework provides the tools you will use to build the web-based GUI that adapter users need to define, deploy, and test their application views. Although each adapter has EIS-specific functionality, all adapters require a GUI for deploying application views. This framework minimizes the effort required to create and deploy this GUI, primarily by using these two components:

- A web application component that allows you to build an HTML-based GUI by using JSPs. This component is augmented by tools such as the JSP templates and tag library and the JavaScript library.
- A deployment helper component that provides a simple API for deploying, undeploying, copying, and editing application views on a WebLogic server.

The design-time interface for each adapter is a J2EE web application that is bundled as a `.war` file. A web application is a bundle of `.jsp`, `.html`, image files, and so on. The web application descriptor is `web.xml`. The descriptor instructs the J2EE web container how to deploy and initialize the web application.

Every web application has a context. The context is specified during deployment and identifies resources associated with the web application under the web container's doc root.

Events and Services

The ADK is used to create two types of adapters: event adapters and service adapters. Within the ADK architecture, services and events are defined as a self-describing objects (that is, the name indicates the business function) that use XML schema to define their input and output.

Events

An event is an XML document published by an application view when an event of interest occurs within an EIS. Clients that want to be notified of events register their interest with an application view. The application view then acts as a broker between the target application and the client. When a client has subscribed to events published by an application view, the application view notifies the client whenever an event of interest occurs within the target application. When an event subscriber is notified that an event of interest has occurred, it is passed an XML document that describes the event. Application views that publish events can also provide clients with the XML schema for the publishable events.

Note: The application view represents a business-level interface to the specific functionality in an application. For more information on this feature, please refer to the *BEA WebLogic Application Integration Product Overview*.

Services

A service is a business operation within an application that is exposed by the application view. It exists as a request/response mechanism; that is, when an application receives a request to invoke a business service, the application view invokes that functionality within its target application and then returns (or, responds with) an XML document that describes the results.

To define a service, you will need to determine and define the input requirements, output expectations, and the content of the interaction specification. A request is submitted in two parts:

- An interaction specification, containing static “secondary metadata” about the request.
- Basic input, which identifies the value of any variables; for example, in a DBMS transaction, the SQL statement is included in the interaction specification and the value of the variable in the input requirement. The result of the transaction is considered the output expectation.

Logging

Logging is an essential feature of an adapter component. Most adapters are used to integrate different applications and do not interact with end users while processing data. Unlike the behavior of a front-end component, when an adapter encounters an error or warning condition, it cannot stop processing and wait for an end-user to respond.

Moreover, the applications that adapters connect to are typically mission-critical business applications. For example, an adapter might be required to keep an audit report of every transaction with an EIS. Consequently, adapter components should provide both accurate logging and auditing information. The ADK's Logging Framework is designed to handle the needs of both logging and auditing.

The Logging Toolkit

The ADK provides the logging toolkit, which allows you to log internationalized messages to multiple output destinations. The logging toolkit leverages the work of the open source project, Apache Log4j.

The logging toolkit wraps the critical classes within log4j to provide added functionality when you are building J2EE-compliant adapters. For the Application Integration product, the log toolkit is provided in the `logtoolkit.jar` file.

For information on using the logging toolkit, see "The ADK Logging Toolkit."

The Logging Framework

With the ADK, logging of adapter activity is accomplished by implementing the logging framework. This framework gives you the ability to log internationalized and localized messages to multiple output destinations. It provides a range of configuration parameters you can use to tailor message category, priority, format, and destination.

The logging framework uses a categorical hierarchy to allow inheritance of logging configuration by all packages and classes within an adapter. The framework allows parameters to be easily modified during runtime.

Internationalization and Localization

The logging framework allows you to internationalize log messages. Internationalized applications are easy to tailor to the idioms and languages of end users around the world without re-factoring the code. Localization is the process of adapting software for a specific region or language by adding locale-specific components and text. The logging framework uses the internationalization and localization facilities provided by the Java platform.

Adapter Logical Name

Each adapter created must have an *adapter logical name*; that is, a unique identifier that represents an individual adapter and serves as the organizing principle for all adapters. As such, the adapter logical name is how an individual adapter is identified and is also used to name the following:

- message bundle
- logging configuration
- log categories

The adapter logical name is a combination of the vendor name, the type of EIS connected to the adapter, and the version number of the EIS. By convention, this information is expressed as *vendor_EIS-type_EIS version*; for example, `BEA_WLS_SAMPLE_ADK`, where:

- `BEA` is the vendor
- `WLS` is the EIS-type
- `SAMPLE_ADK` is the EIS version

CHAPTER

3 Tools

The ADK provides a robust set of tools to assist you in developing adapters and the design-time GUI.

This section includes information on the following subjects:

- Sample Adapter
- The GenerateAdapterTemplate Utility
- ADK Javadoc
- Ant-Based Build Process
- XML Tools

Sample Adapter

The ADK contains a sample adapter that provides non-EIS specific code examples to help you start building an adapter. (Do not confuse this sample adapter with the eMail and DBMS example adapters also included with Application Integration; these example adapters are documented in “The DBMS Sample Adapter” and “The eMail Sample Adapter”). You can find the sample adapter in `<WLAI_HOME>/dev/sample`.

Why Use the Sample Adapter?

The purpose of the sample adapter is to free you from much of the coding necessary to build an adapter. It provides concrete implementations of key abstract classes that only require customization for your specific EIS. In addition, the ADK provides the GenerateAdapterTemplate utility with which you can quickly clone the sample adapter development tree for use by the adapter you are developing. See "The GenerateAdapterTemplate Utility."

What's In the Sample Adapter

Specifically the sample adapter contains:

`sample.spi.ManagedConnectionFactoryImpl`
: A concrete extension to `AbstractManagedConnectionFactory` that you can customize for a specific EIS.

`sample.spi.ManagedConnectionImpl`
: A concrete extension to `AbstractManagedConnection` that you can customize this class for a specific EIS.

`sample.spi.ConnectionMetaDataImpl`
: A concrete extension to `AbstractConnectionMetaData` that you can customize for a specific EIS.

`sample.web.DesignTimeRequestHandler`
: A concrete extension to `AbstractDesignTimeRequestHandler` that shows how to add an event or service at designtime.

`sample.event.EventGenerator`
: A concrete extension to `AbstractPullEventGenerator` that shows how to extend the ADK base class to construct an event generator.

Note: For more details on the classes extended by those in the sample adapter, please refer to the ADK Javadocs.

The GenerateAdapterTemplate Utility

To facilitate using the sample adapter, the ADK provides GenerateAdapterTemplate, a command-line utility you can use to clone the sample adapter development tree and create a new adapter development tree. See “Creating a Custom Development Environment” for complete instructions on using this tool.

ADK Javadoc

ADK classes, interfaces, methods, and constructors are defined in the development kit’s Javadocs. Javadocs are included with the ADK installation and are stored in `<WLAI_HOME>\dev\doc\api`.

Ant-Based Build Process

The ADK employs a build process based upon Ant, a 100% pure Java-based build tool. For the ADK, Ant does the following:

- Creates a Java archive (`.jar`) file for the adapter.
- Creates a `.war` file for an adapter’s web application.
- Creates a `.rar` file for a J2EE-compliant adapter.

Why Use Ant?

Traditionally, build tools are inherently shell-based. They evaluate a set of dependencies and then execute commands, not unlike those you would issue on a shell. While it is simple to extend these tools by using or writing any program for your operating system, you are also limited to that OS, or at least that OS type (for example, Unix).

Ant is preferable to shell-based make tools for the following reasons:

- Instead of a model where it is extended with shell-based commands, it is extended using Java classes.
- Instead of writing shell commands, the configuration files are XML-based, calling out a target tree where various tasks get executed. Each task is run by an object that implements a particular task interface. While this removes some of the expressive power inherent in being able to construct a shell command, it gives your application the ability to be cross-platform.
- If you want to execute a shell command, Ant has an execute rule that allows different commands to be executed based on the OS upon which it executing.

For complete instruction on setting up Ant, see “Step 2c: Setting Up the Build Process.”

XML Tools

The ADK ships with two XML development tools. While not part of the design-time component, these tools are required for building a design-time GUI. These tools are considered part of the metadata support layer for the design-time framework. These tools are:

- Schema Object Model (SOM), which is an API for programmatically building XML schemas. An adapter calls into an EIS for specific request/response metadata, which then needs to be programatically transformed into an XML schema. The SOM is a set of tools that extracts many of the common details, such as syntactical complexities of schema so that you can focus on its more fundamental aspects.

- IDocument, which provides the x-path interface to a document object model (DOM) document.

For more information on both of these tools, please refer to their respective Javadocs:

- For SOM, go to `<WLAI_HOME>\dev\doc\com\bea\schema`.
- For iDocument, go to `<WLAI_HOME>\dev\doc\com\bea\document`.

CHAPTER

4 Creating a Custom Development Environment

Warning: We strongly recommend that you *do not* directly alter the sample adapter included with the ADK. Instead, use the GenerateAdapterTemplate utility described in this chapter. Modifying the sample adapter by any other means might result in unexpected and unsupported behavior.

To facilitate using the sample adapter (see "Sample Adapter"), the ADK provides GenerateAdapterTemplate, a command-line utility you can use to clone the sample adapter development tree and create a new adapter development tree.

This section provides information on the following subjects:

- Adapter Setup Worksheet
- Using GenerateAdapterTemplate
  - Step 1. Execute GenerateAdapterTemplate
  - Step 2. Rebuild the Tree
  - Step 3. Deploy the Adapter to Application Integration

Adapter Setup Worksheet

The adapter setup worksheet is a questionnaire that will help you identify and collect critical information about the adapter you are developing. You can find this questionnaire in "Adapter Setup Worksheet."

This worksheet is a set of 20 questions that will help you identify critical adapter information, such as EIS type, vendor, and version, locale and national language of the deployment, the adapter logical name, and whether or not the adapter will supports services. When you run GenerateAdapterTemplate, you will be prompted to enter this information. When the information is processed, a custom development tree for your adapter will be created.

Using GenerateAdapterTemplate

This section describes how to use GenerateAdapterTemplate. You will need to perform the following steps:

- Step 1. Execute GenerateAdapterTemplate
- Step 2. Rebuild the Tree
- Step 3. Deploy the Adapter to Application Integration

Step 1. Execute GenerateAdapterTemplate

To use this tool, do the following:

1. Open a command-line from the `<WLAI_HOME>/dev/bin` directory and execute one the following commands:
   - For Windows NT: `GenerateAdapterTemplate.cmd`
   - For Unix: `GenerateAdapterTemplate.sh`

   The system responds:

```
<WLAI_HOME>\dev\bin>generateadaptertemplate
************************************************************
Welcome! This program helps you generate a new adapter
development tree by cloning the ADK's sample adapter development
tree.
Do you wish to continue? (yes or no); default='yes':
```

2. Select yes by pressing Enter.

   The system responds:

   ```
   Please choose a name for the root directory of your adapter
   development tree:
   ```

3. Enter a unique, easy-to-remember directory name (for example *dir_name*) and press Enter.

   The system responds:

   ```
   created directory d:\wlai\dev\dir_name
   ```

   (where *dir_name* is the new directory name.)

   Note: If you entered a directory name that already exists, the system will respond:

   ```
   d:\wlai\dev\sample already exists, please choose a new
   directory that does not already exist!
   Please choose a name for the root directory of your adapter
   development tree:
   ```

   ```
   Enter the EIS type for your adapter:
   ```

4. Enter an identifier for the EIS type to which your adapter will be connecting. Press Enter.

   The system responds:

   ```
   Enter a short description for your adapter:
   ```

5. Enter a short, meaningful description of the adapter you are about to develop and press Enter.

   The system responds:

   ```
   Enter the major version number for your adapter; default='1':
   ```

6. Either press Enter to accept the default or enter the appropriate version number and then press Enter.

The system responds:

```
Enter the minor version number for your adapter; default='0':
```

7. Either press Enter to accept the default or type the appropriate minor version number and then press Enter.

   The system responds:

   ```
   Enter the vendor name for your adapter:
   ```

8. Enter the vendor's name and press Enter.

   The system responds:

   ```
   Enter an adapter logical name; default='default_name':
   ```

9. Either press Enter to accept the default or type the adapter logical name you want to use. Press Enter. The default adapter logical name (*'default_name'*) is based upon the Application Integration recommended format of *vendor name_EIS-type_version-number*.

   The system responds:

   ```
   Enter the Java package base name for your adapter (e.g. sample
   adapter's is sample): Java package base name
   ```

10. Enter the Java package base name in package format and press Enter. Package format is dot-separated and begins with your URL extension (`.com`, `.org`, `.edu`, and so on), followed by the company name, then by additional adapter identifiers; for example, `com.your_co.adapter.EIS`.

    The system responds:

    ```
    The following information will be used to generate your new
    adapter development environment:
    EIS Type = 'SAP R/3'
    Description = 'description'
    Major Version = '1'
    Minor Version = '0'
    Vendor = 'vendor_name'
    Adapter Logical Name = 'adapter_logical_name'
    Java Package Base = 'com.java.package.base'
    Are you satisfied with these values? (enter yes or no or q to
    quit);
    ```

```
default='yes':
```

11. To confirm the information, press Enter.

    The system responds with the appropriate build information.

Note: If you enter `no`, you will be routed back to Step 4. If you enter q (quit), the application will terminate.

Step 2. Rebuild the Tree

After completing the clone process, change to the new directory and use Ant, the ADK's build tool to rebuild the entire tree. For more information on Ant, see "Ant-Based Build Process" in "Tools."

To rebuild the tree by using Ant, do the following:

1. Edit `antEnv.cmd` (Windows) or `antEnv.sh` (UNIX) in `<WLAI_HOME>\dev\project`.
2. Set the following variables to valid paths:
   - `BEA_HOME` - The location of your BEA_HOME directory.
   - `WLAI_HOME` - The location of your Application Integration installation.
   - `JAVA_HOME` - The location of your Java Development Kit.
   - `WL_HOME` - The location of your WebLogic 6.0 installation.
   - `ANT_HOME` - The location of your Ant home, typically `<WLAI_HOME>\dev\project`.

Notes: The installer will perform this step for you; however, you should be aware that these settings control the Ant process.

On UNIX, the Ant file in `<WLAI_HOME>\dev\project\ant` needs to have an execute permission set. To add the execute permission, type `chmod u+x ant`.

3. Execute `antEnv` from the command-line to set the necessary environment variables for your shell.

4. Execute `ant release` from the `<WLAI_HOME>\dev\<CLONE>\project` directory to build the adapter (`<CLONE>` is the name of the new adapter development root).

   Executing `ant release` will generate the Javadoc for the adapter. You can view the Javadoc by going to:

   `<WLAI_HOME>/dev/<CLONE>/src/overview.html`

   This file provides environment specific instructions for deploying your adapter in Application Integration. Specifically, it provides `config.xml` entries and the replacements for the path already made. In addition, the file provides mapping information. You can copy the contents of `overview.html` directly into `config.xml`, which will facilitate adapter deployment, as described in "Step 3. Deploy the Adapter to Application Integration."

Step 3. Deploy the Adapter to Application Integration

After rebuilding the new adapter, deploy it into Application Integration. To do so, follow these three basic steps:

Note: For more detailed information on adapter deployment, see "Deploying the Adapter" in either "Developing a Service Adapter" or "Developing an Event Adapter."

1. First, declare the adapter's `.rar`, `.war`, and event router `.war` files in your domain's `config.xml` file. You will need to have the XML elements described in Listing 4-1 in your `config.xml` file:

   Note: You can simply copy the code shown in Listing 4-1 from `<WLAI_HOME>/dev/<CLONE>/src/overview.html` into `config.xml`.

Listing 4-1 Sample Code for Deploying an Adapter Created with GenerateAdapterTemplate

```
<!-- This deploys the WebLogic Specific .rar file -->
<Application
     Deployed="true"
     Name="BEA_WLS_SAMPLE_ADK_WLS.rar.rar"
```

```
      Path="d:\bea\wlai1.0\dev\sample\lib">
  <ResourceAdapterComponent
      Name="BEA_WLS_SAMPLE_ADK_WLS.rar"
      Targets="myserver"
      URI="BEA_WLS_SAMPLE_ADK_WLS.rar.rar"/>
</Application>
<!-- This deploys the Web application for design time -->
<Application
      Deployed="true"
      Name="BEA_WLS_SAMPLE_ADK_Web"
      Path="d:\bea\wlai1.0\dev\sample\lib">
  <WebAppComponent
      Name="BEA_WLS_SAMPLE_ADK_Web"
      ServletReloadCheckSecs="1"
      Targets="myserver"
      URI="BEA_WLS_SAMPLE_ADK_Web"/>
</Application>
<!-- This deploys the event router Web application -->
<Application
      Deployed="true"
      Name="BEA_WLS_SAMPLE_ADK_EventRouter"
      Path="d:\bea\wlai1.0\dev\sample\lib">
  <WebAppComponent
    Name="BEA_WLS_SAMPLE_ADK_EventRouter"
    Targets="myserver"
    URI="BEA_WLS_SAMPLE_ADK_EventRouter.war"/>
</Application>
```

Note: Replace all occurrences of `<WLAI_HOME>\dev\sample\lib` with the correct path to the adapter's `lib` directory for your environment in all elements shown in Listing 4-1.

2. Provide the mapping of the adapter deployment to the adapter's design-time web application. To do this, use this procedure:

   a. Make an entry into `<WLAI_DEPLOY_HOME>\wlai.properties` to map the path to the design-time web application. `<WLAI_DEPLOY_HOME>` is the Application Integration directory under your domain.

   b. Add a property for each adapter you will deploy. The property will point at the design-time interface web application. This should include only the path portion of the URL for the web application (not the full server name and port, etc.).

   c. Optionally, supply only the web application context (usually the name of the `.war` file you place in the application's directory) and omit the `/display.jsp` or equivalent. The property should be of the form:

      ```
      wlai.adapter.<Vendor name from .rar>.<Display name from
      .rar>.webAppPath=<web app path>
      ```

      `<Vendor name from .rar>` and `<Display name from .rar>` cannot contain the characters "." (dot), "=" (equals), or a blank space. Replace these characters by using the following mappings:

      - Replace all instances of "." with "+" (plus)
      - Replace all blank spaces with "_" (underscore)
      - Replace all instances of "=" with "-" (dash)

      For this adapter, the entry should be:

      ```
      wlai.adapter.__BEA__.BEA_WLS_SAMPLE_ADK.webAppPath=
      BEA_WLS_SAMPLE_ADK_Web
      ```

3. Next, add the `.jar` file(s) for the new adapter to the WebLogic server `CLASSPATH`. The reason for this is due to a limitation in WebLogic 6.0 regarding enterprise applications. Ideally, using an `.ear` file would be a viable solution; however, the enterprise application facility in WebLogic Server 6.0 SP2 does not fully support the sharing of a `.jar` between a `.war` and `.rar`. The result of putting these files in an `.ear` file and taking the `.jar` out of the `CLASSPATH` is

that you can't test events or services at designtime. So, in order to be able to test, you must put the `.jar` in the `CLASSPATH`, which renders the creation of an `.ear` useless.

For this version of the ADK, you should add the `.jar` file for your adapter to the `ADAPTER_CLASSPATH` in `wlaiEnv.cmd/sh`, as shown in Listing 4-2:

Listing 4-2 Sample Code Showing How to Add the .jar File to a Classpath

```
set ADAPTER_CLASSPATH=%WLAI_LIB_HOME%\logtoolkit.jar;
%WLAI_LIB_HOME%\adk.jar;%WLAI_HOME%\dev\sample\lib\BEA_WLS_SAMP
LE_ADK.jar;%WLAI_HOME%\dev\dbms\lib\BEA_WLS_DBMS_ADK.jar;%WLAI_
HOME%\dev\email\lib\BEA_WLS_EMAIL_ADK.jar
```

4. Go to the Application View Management Console at:

   `http://<host>:<port>/wlai`

   and attempt to add a new application view for your adapter. This will verify that the adapter has been deployed correctly. For instructions on adding an application view, see "Defining Application Views" in the *BEA WebLogic Application Integration User Guide*.

5. Restart WebLogic 6.0.

CHAPTER

5 The ADK Logging Toolkit

Logging is an essential feature of an adapter component. Most adapters are used to integrate different applications and do not interact with end users while processing data. Unlike the behavior of a front-end component, when an adapter encounters an error or warning condition, it cannot stop processing and wait for an end-user to respond.

With the ADK, logging adapter activity is accomplished by implementing a logging framework. This framework gives you the ability to log internationalized and localized messages to multiple output destinations. It provides a range of configuration parameters you can use to tailor message category, priority, format, and destination.

This section contains information on the following subjects:

- Logging Toolkit
- Logging Configuration File
- Logging Concepts
  - Message Categories
  - Message Priority
  - Message Appenders
  - Message Layout
  - Putting the Components Together
- How to Set Up Logging
- Logging Framework Classes

  - `com.bea.logging.ILogger`
  - `com.bea.logging.LogContext`
  - `com.bea.logging.LogManager`
- Internationalization and Localization of Log Messages
- Saving Contextual Information in a Multi-Threaded Component

Logging Toolkit

The ADK's logging toolkit allows you to log internationalized messages to multiple output destinations. The logging toolkit leverages the work of the open source project Apache Log4j. This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

The logging toolkit is a container that wraps the necessary Log4j classes to provide added functionality for J2EE-compliant adapters. It is provided in the `logtoolkit.jar` file under: `<WLAI_ROOT>/lib`. This `.jar` file depends on DOM, XERCES, and Log4j 1.1.2. The XERCES dependency is satisfied by `weblogic.jar` and `xmlx.jar` provided in the WebLogic 6.0 distribution (`<WL_HOME>/lib`). The Application Integration distribution includes the required version of Log4j in (`<WLAI_HOME>/resources/Log4j`).

The Log4j package is distributed under the Apache Public License, a full-fledged open source license certified by the open source initiative. The latest Log4j version, including full-source code, class files and documentation can be found at the Apache Log4j Web site.

Logging Configuration File

Throughout this section, you will see references to and code snippets from the logging configuration file. This file is a `.xml` file that is identified by the adapter logical name; for example, `BEA_WLS_DBMS_ADK.xml`. It contains the base information for the four logging concepts discussed in "Logging Concepts" and can be modified for your specific adapter.

The ADK provides a basic logging configuration file, `BEA_WLS_SAMPLE_ADK.xml`, in `<WLAI_HOME>/dev/sample/src`. To modify this file for your adapter, run GenerateAdapterTemplate. This utility will customize the sample version of the logging configuration file with information pertinent to your new adapter and place the customized version in the new adapter's development environment. For more information on GenerateAdapterTemplate, see "Creating a Custom Development Environment."

Logging Concepts

Prior to using the logging toolkit provided with the ADK, you should understand a few key concepts of the logging framework. Logging has four main components:

- Message Categories
- Message Priority
- Message Appenders
- Message Layout

These components work together to enable you to log messages according to message type and priority, and to control at runtime how these messages are formatted and where they are reported.

Message Categories

Categories identify log messages according to criteria you defined and are a central concept of the logging framework. In the ADK, a category is identified by its name, such as `BEA_WLS_SAMPLE_ADK.DesignTime`.

Categories are hierarchically defined. That is, any category can inherit properties from parent categories. The hierarchy is defined thusly:

- A category is an ancestor of another category if its name followed by a dot is a prefix of the descendant category name.
- A category is a parent of a child category if there are no ancestors between itself and the descendant category.

For example, `BEA_WLS_SAMPLE_ADK.DesignTime` is a descendant of `BEA_WLS_SAMPLE_ADK`, which is a descendant of the root category. For example:

```
ROOT CATEGORY
  |
  |->BEA_WLS_SAMPLE_ADK
       |
       |->BEA_WLA_SAMPLE.ADK.DesignTime
```

The root category resides at the top of the category hierarchy; it always exists and it cannot be retrieved by name.

When you create categories, you should name them according to components in their adapter. For example, if an adapter has a design-time user interface component, the adapter could have a category, `BEA_WLS_SAMPLE_ADK.DesignTime`.

Message Priority

Every message has a priority that indicates its importance. Message priority is determined by the method on the ILogger interface used to log the message. In other words, calling the debug method on an ILogger instance generates a debug message.

The logging toolkit supports five possible priorities for a given message, as described in Table 5-1:

Table 5-1 Logging Toolkit Priorities

Priority	Description
AUDIT	Indicates an extremely important log message that relates to the business processing performed by an adapter. Messages with this priority will always be written to the log output.
ERROR	Indicates an error in the adapter. Error messages are internationalized/localized for the user.
WARN	Indicates a situation that is not an error, but could cause problems in the adapter. A warning message that is internationalized/localized for the user.
INFO	Indicates an informational message that is internationalized/localized for the user.
DEBUG	Indicates a debug message, which are used to determine how the internals of a component are working and are typically not internationalized.

The BEA_WLS_SAMPLE_ADK category has priority DEBUG because of the following child element:

```
<priority value='DEBUG' class='com.bea.logging.LogPriority'/>
```

The class for the priority must be com.bea.logging.LogPriority.

Assigning a Priority to a Category

You can assign a priority to a Category. If a given category is not assigned a priority, it inherits one from its closest ancestor with an assigned priority; that is, the inherited priority for a given category is equal to the first non-null priority in the category hierarchy, starting at the given category and proceeding upwards in the hierarchy towards the root category.

A log message will be output to the log destination if its priority is higher than or equal to the priority of its category. Otherwise, the message will not be written to the log destinations. A category without an assigned priority will inherit one from the hierarchy. To ensure that all categories can eventually inherit a priority, the root

category always has an assigned priority. A log statement of priority p in a category with inherited priority q, is enabled if p >= q. This rule assumes that priorities are ordered as follows: `DEBUG` < `INFO` < `WARN` < `ERROR` < `AUDIT`.

Message Appenders

The logging framework allows an adapter to log to multiple destinations by using an interface called an appender. Log4j provides appenders for:

- The console
- Files
- Remote socket servers
- NT Event Loggers
- Remote UNIX Syslog daemons

In addition, the ADK log toolkit provides a WebLogic appender that you can specify to output the log message to your WebLogic Server log.

A category may refer to multiple appenders. Each enabled logging request for a given category will be forwarded to all the appenders in that category, as well as the appenders higher in the hierarchy. In other words, appenders are inherited additively from the category hierarchy. For example, if a console appender is added to the root category, then all enabled logging requests will at least print on the console. If in addition a file appender is added to category "C," then enabled logging requests for C and C's children will print to a file and on the console. It is possible to override this default behavior so that appender accumulation is no longer additive by setting the additivity flag to false.

Note: If you've also added the console appender to directly to C, you will get two messages—one from C and one from the root—on the console. This is because the root category always logs to the console.

Listing 5-1 shows an appender for the WebLogic log:

Listing 5-1 Sample Code Showing an Appender for the WebLogic Log

```
<!--

  A WeblogicAppender sends log output to the WebLogic log.

  It can only be used if you are running inside WebLogic.

-->

<appender name="WebLogicAppender"
class="com.bea.logging.WeblogicAppender">

      <!-- Use a basic LOG4J pattern layout -->

  <layout class="org.apache.Log4j.PatternLayout">

    <param name="ConversionPattern" value="%-5p %d{DATE} %c{4} %x
- %m%n"/>

  </layout>

  </appender>
```

Message Layout

By using Log4j, you can also customize the format of a log message. This is accomplished by associating a layout with an appender. The layout is responsible for formatting a log message while an appender directs the formatted message to its destination. The log toolkit typically uses the PatternLayout to format its log messages. The PatternLayout, part of the standard Log4j distribution, lets you specify the output format according to conversion patterns similar to the C language `printf` function.

For example, the PatternLayout with the conversion pattern `%-5p%d{DATE} %c{4} %x - %m%n` will output a message like:

```
AUDIT 21 May 2001 11:00:57,109 BEA_WLS_SAMPLE_ADK - admin opened
connection to EIS
```

In the pattern,

- `%-5p` displays the priority of the message; in the example shown above, this is `AUDIT`

- %d{DATE} displays the date of the message; in the example shown above, this is 21 May 2001 11:00:57,109
- %c{4} displays the category for the log message; in the example shown above, this is BEA_WLS_SAMPLE_ADK

The text after the "-" is the message of the statement.

Putting the Components Together

Listing 5-2 declares a new category for the sample adapter, sets its priority, and declares an appender to output log messages to a file:

Listing 5-2 Sample XML Code for Declaring a New Log Category

```
<!-

IMPORTANT!!! ROOT Category for the adapter; making this unique
prevents other adapters from logging to your category

-->

<category name='BEA_WLS_SAMPLE_ADK' class='com.bea.logging.
LogCategory'>

  <!-

    Default Priority Level; may be changed at runtime

    DEBUG means log all messages from the adapter's code base

    INFO means log informationals, warnings, errors, and audits

    WARN means log warnings, errors, and audits

    ERROR means log errors and audits

    AUDIT means log audits only

  -->

  <priority value='DEBUG' class='com.bea.logging.LogPriority'/>

  <!--

   Associate one or more output destinations with this
```

```
   category; in this case, the appender is set in an <appender>
   element called 'FileAppender'
  -->
  <appender-ref ref='FileAppender'/>
</category>
```

Note: You must specify the class as `com.bea.logging.LogCategory`.

How to Set Up Logging

Note: The following procedure assumes that you've cloned a development environment by running the GenerateAdapterTemplate utility. For more information on this utility, see "Creating a Custom Development Environment."

Setting up the logging framework for your adapter is a four-step process.

1. Identify all of the basic components used in the adapter. For example, if your adapter has an EventGenerator, you might want to have an EventGenerator component; if it supports a design-time GUI, you will need a design-time component.
2. Open the base log configuration file from the cloned adapter. This file is found in `<WLAI_HOME>/dev/<CLONE>/src/` and will have the extension `.xml`. For example, the DBMS adapter configuration file is in `<WLAI_HOME>/dev/dbms/src`. It's called `BEA_WLS_DBMS_ADK.xml`.
3. In the base log configuration file, add the category elements for all adapter components you identified. For each category element, establish a priority. Listing 5-3 shows how a category for an EventGenerator with a priority of DEBUG is added.

Listing 5-3 Sample Code Adding an EventGenerator Log Category with a Priority of DEBUG

```
<category name='BEA_WLS_DBMS_ADK.EventGenerator'
          class='com.bea.logging.LogCategory'>
  <priority value='DEBUG'
            class='com.bea.logging.LogPriority'/>
</category>
```

4. Determine the appender and add it to the configuration file. If necessary, add message formatting information. Listing 5-4 shows how a basic file appender is added within the `<appender>` element. Instructions within the `<layout>` element identify the message format pattern.

Listing 5-4 Sample Code Adding a File Appender and Layout Pattern

```
<!-- A basic file appender -->
<appender name='FileAppender'
class='org.apache.Log4j.FileAppender'>
    <!-- Send output to a file -->
    <param name='File' value='BEA_WLS_DBMS_ADK.log'/>
    <!-- Truncate existing -->
    <param name="Append" value="true"/>
    <!-- Use a basic LOG4J pattern layout -->
    <layout class='org.apache.Log4j.PatternLayout'>
     <param name='ConversionPattern' value='%-5p %d{DATE} %c{4}
%x - %m%n'/>
    </layout>
</appender>
```

At this point, you should review these other configuration files to confirm their settings.

- `<CLONE>/src/eventrouter/web-inf/web.xml`; The AbstractEventGenerator uses the logging information entered in the base configuration file to configure the log framework at initialization time.
- `<CLONE>/src/rar/META-INF/ra.xml and weblogic-ra.xml`; The AbstractManagedConnectionFactory uses the logging information entered in the base configuration file to configure the log framework at initialization time.
- `<CLONE>/src/war/web-inf/web.xml`; The RequestHandler (the parent of AbstractDesignTimeRequestHandler) uses the logging information entered in the base configuration file to configure the log framework at initialization time.

Logging Framework Classes

In addition to understanding the basic concepts of the logging framework, you will also need to understand three main classes provided in the log toolkit:

- `com.bea.logging.ILogger`
- `com.bea.logging.LogContext`
- `com.bea.logging.LogContext`

com.bea.logging.ILogger

This is the main interface to the logging framework. It provides numerous convenience methods for logging messages.

In "How to Set Up Logging," you saw how you can configure logging in the base log configuration file. You can also configure logging programmatically by implementing the logging methods listed below:

- `logger.setPriority("DEBUG");` changes the minimum priority of messages printed from the current ILogger.
- `logger.addRuntimeDestination (writer);` adds an additional appender used when the container passes its PrintWriter to the adapter.

- `logger.warn("Some message", true);` logs a message with the priority level WARN, without using the ResourceBundle. The boolean indicates that the string is a message, not a key.
- `logger.warn("someKey");` logs a message with the priority level WARN, by looking it up with "`someKey`" in the ResourceBundle.
- `logger.info("someKey", anObjArray);` logs a message with the priority level INFO by looking up a template with `someKey` in the ResourceBundle and filling in the blanks with the elements of `anObjArray`.
- `logger.error(exception);` logs a message with the priority level `ERROR`, by passing an exception (Throwable) to this method. It will call `getMessage()`, and include a stack trace. All logging methods that take a Throwable as an argument log a stack trace.

com.bea.logging.LogContext

This class encapsulates information needed to identify an ILogger instance in the logging framework. Currently, a LogContext encapsulates a log category name and a locale, such as `en_US`.

com.bea.logging.LogManager

This class provides a method to allow you to configure the logging framework and provides access to ILogger instances.

To properly configure the log toolkit for your adapter, the ADK implements the LogManager's `configure()` method with the arguments shown in Listing 5-5:

Listing 5-5 Sample Code for Configuring the Log Toolkit

```
public static LogContext
  configure(String strLogConfigFile,
            String strRootLogContext,
```

```
String strMessageBundleBase,
Locale locale,
ClassLoader classLoader)
```

Table 5-2 describes the arguments passed by `configure()`:

Table 5-2

Argument	Description
`strLogConfigFile`	This file contains the log configuration information for your adapter. The file should exist on the classpath. We recommend that you include this file into your adapter's main `.jar` file so that it can be included in the `.war` and `.rar` files for your adapter. This file should conform to the `Log4j.dtd`. The `Log4j.dtd` file is provided in the `Log4j.jar` in the Application Integration distribution.
`strRootLogContext`	This is the name of the logical root of the category hierarchy for your adapter. For the sample adapter, this value is `BEA_WLS_SAMPLE_ADK`.
`strMessageBundleBase`	This is the base name for the message bundles for your adapter. It is required by the ADK that you use message bundles. For the sample adapter, this value is `BEA_WLS_SAMPLE_ADK`.
`locale`	This identifies the locale (language and nation). The log toolkit organizes categories into different hierarchies based on locale. For example, if your adapter supports two locales `en_US` and `fr_CA`, the log toolkit will maintain two category hierarchies, one for `en_US` and one for `fr_CA`.
`classLoader`	This is the ClassLoader the LogManager should use to load resources, such as ResourceBundles and log configuration files.

Once the configuration is complete, you can retrieve ILogger instances for your adapter by supplying a LogContext object:

Listing 5-6 Sample Code for Supplying a LogContext Object

```
LogContext logContext = new LogContext("BEA_WLS_SAMPLE_ADK",
java.util.Locale.US);

ILogger logger = LogManager.getLogger(logContext);

logger.debug("I'm logging now!");
```

The ADK hides most of the log configuration and setup for you. The `com.bea.adapter.spi.AbstractManagedConnectionFactory` class configures the log toolkit for service adapters and the AbstractEventGenerator configures the log toolkit for event adapters. In addition, all of the CCI and SPI base classes provided in the ADK provide access to an ILogger and its associated LogContext.

For other layers in the adapter to access the correct ILogger object, there are two approaches you can take.

Note: "Other layers" refers to layers in an adapter that support the CCI/SPI layer, such as a socket layer for establishing communication to the EIS.

- **Approach 1**: The CCI/SPI layers can pass the `LogContext` object into the lower layers. This works but also adds overhead.
- **Approach 2**: The CCI layer can establish the `LogContext` for the current running thread at the earliest possible place in the code. The ADK's `com.bea.adapter.cci.ConnectionFactoryImpl` class sets the `LogContext` for the current running thread in the `getConnection()` methods. The `getConnection()` methods are the first point of contact a client program has with your adapter. Consequently, lower layers in an adapter can safely access the LogContext for the current running thread using the following code:

  ```
  public static LogContext getLogContext(Thread t)
     throws IllegalStateException, IllegalArgumentException
  ```

  Additionally, we supply a convenience method on the LogManager:

  ```
  public static ILogger getLogger() throws IllegalStateException
  ```

  This method provides an ILogger for the current running thread. There is one caveat to using this approach, lower layers should not store the LogContext or ILogger as members. Rather, they should dynamically retrieve them from the

LogManager. An IllegalStateException is thrown if this method is called before a LogContext is set for the current running thread.

Internationalization and Localization of Log Messages

Internationalization and Localization are central concepts to the ADK logging framework. All logging convenience methods on the ILogger interface, except the debug methods, allow internationalization. The implementation follows the Java Internationalization standards, using ResourceBundle objects to store locale-specific messages or templates. Sun provides a good online tutorial on using the I18N ("internationalization") and L10N ("localization") standards of the Java language.

Saving Contextual Information in a Multi-Threaded Component

Most real-world systems have to deal with multiple clients simultaneously. In a typical multi-threaded implementation of such a system, different threads will handle different clients. Logging is especially well suited to trace and debug complex distributed applications. A common approach to differentiate the logging output of one client from another is to instantiate a new separate category for each client. This promotes the proliferation of categories and increases the management overhead of logging.

A lighter technique is to uniquely stamp each log request initiated from the same client interaction. Neil Harrison described this method in "Patterns for Logging Diagnostic Messages," in *Pattern Languages of Program Design 3*, edited by R. Martin, D. Riehle, and F. Buschmann (Addison-Wesley, 1997).

To uniquely stamp each request, the user pushes contextual information into the Nested Diagnostic Context (NDC). The log toolkit provides a separate interface for accessing NDC methods. The interface is retrieved from the ILogger by using the method `getNDCInterface()`.

NDC printing is turned on in the XML configuration file (with the symbol %x). Every time a log request is made, the appropriate logging framework component includes the entire NDC stack for the current thread in the log output. The user will not need to intervene in this process. In fact, the user is responsible only for placing the correct information in the NDC by using the push and pop methods at a few well-defined points in the code.

Listing 5-7 Sample Code

```
public void someAdapterMethod(String aClient) {
  ILogger logger = getLogger();
  INestedDiagnosticContext ndc = logger.getNDCInterface();
  // I'm keeping track of this client name for all log messages
  ndc.push("User name=" + aClient);
  // method body …
  ndc.pop();
}
```

A good place to use the NDC is in your adapter's CCI Interaction object.

CHAPTER

6 Developing a Service Adapter

Service adapters receive an XML request document from a client and invoke the associated function in the underlying EIS. They are consumers of messages and may or may not provide a response. Service adapters perform the following four functions:

- They receive service requests from an external client.
- They transform the XML request document into the EIS specific format. The request document conforms to the request XML schema for the service. The request XML schema is based on metadata in the EIS.
- They invoke the underlying function in the EIS and wait for its response.
- They transform the response from the EIS specific data format to an XML document that conforms to the response XML schema for the service. The response XML schema is based on metadata in the EIS.

This section contains information on the following subjects:

- Service Adapters in the Runtime Environment
- The Flow of Events
- Step 1: Development Considerations
- Step 2: Configuring the Development Environment
  - Step 2a: Set Up the File Structure
  - Step 2b: Assign the Adapter Logical Name
  - Step 2c: Setting Up the Build Process

- Step 6: Testing the Adapter
  - Using the Test Harness
  - Test Case Extensions Provided by the ADK

Service Adapters in the Runtime Environment

Figure 6-1 and Figure 6-2 show the processes executed when a service adapter is used in the runtime environment. Figure 6-1shows an asynchronous service adapter while Figure 6-2 shows a synchronous adapter.

Figure 6-1 An Asynchronous Service Adapter in the Runtime Environment

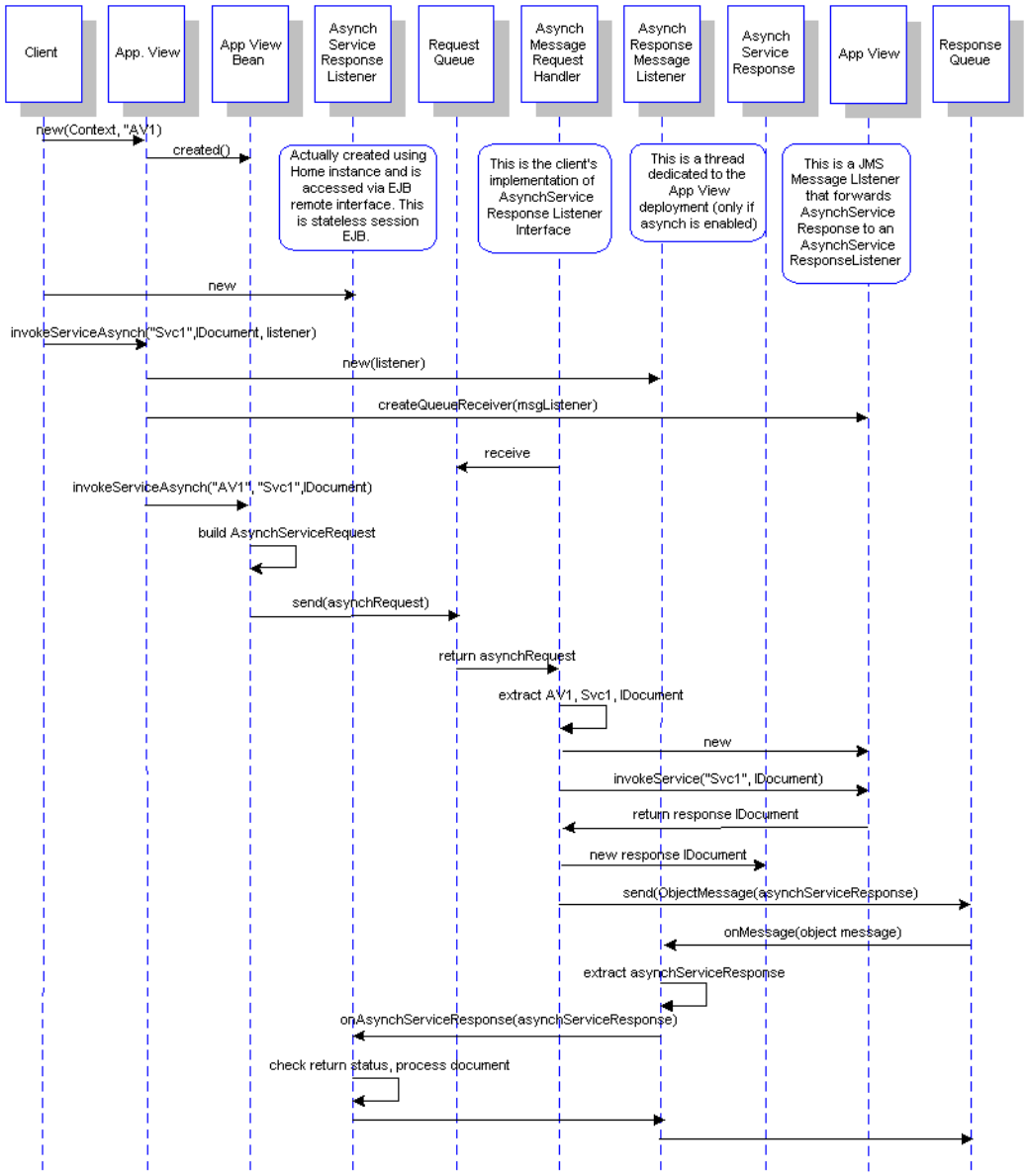
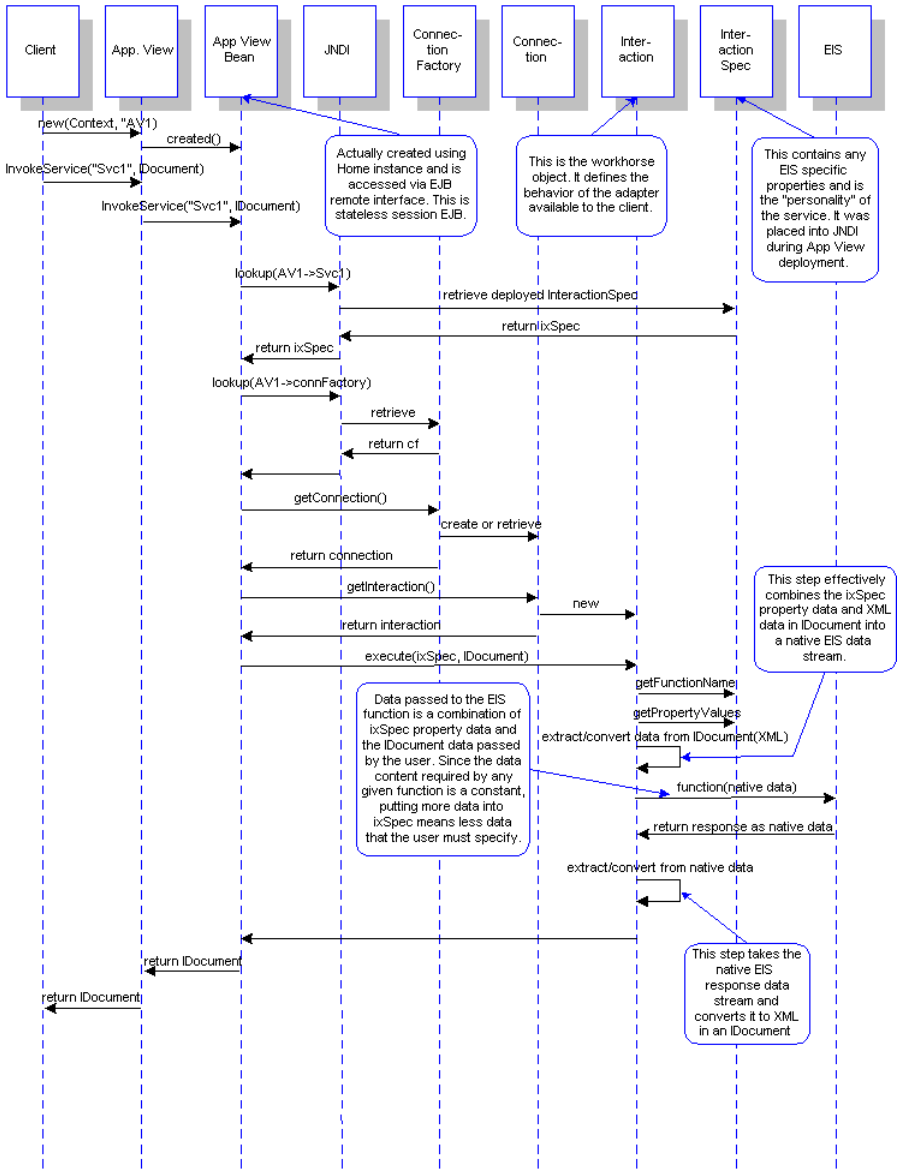


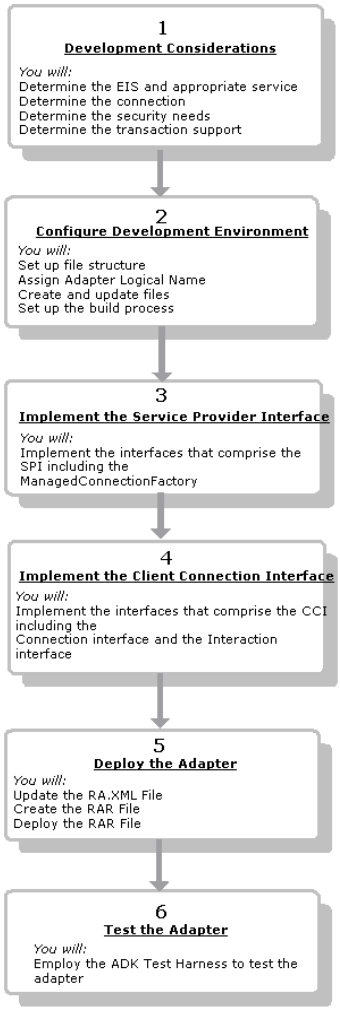
Figure 6-2 A Synchronous Service Adapter in the Runtime Environment


Client
App. View
App View Bean
JNDI
Connec-tion Factory
Connec-tion
Inter-action
Inter-action Spec
EIS
new(Context, "AV1")
created()
InvokeService("Svc1", Document)
InvokeService("Svc1", IDocument)
Actually created using Home instance and is accessed via EJB remote interface. This is stateless session EJB.
This is the workhorse object. It defines the behavior of the adapter available to the client.
This contains any EIS specific properties and is the "personality" of the service. It was placed into JNDI during App View deployment.
lookup(AV1->Svc1)
retrieve deployed InteractionSpec
returnIxSpec
return ixSpec
lookup(AV1->connFactory)
retrieve
return cf
getConnection()
create or retrieve
return connection
getInteraction()
new
return interaction
execute(ixSpec, IDocument)
This step effectively combines the ixSpec property data and XML data in IDocument into a native EIS data stream.
getFunctionName
getPropertyValues
extract/convert data from IDocument(XML)
Data passed to the EIS function is a combination of ixSpec property data and the IDocument data passed by the user. Since the data content required by any given function is a constant, putting more data into ixSpec means less data that the user must specify.
function(native data)
return response as native data
extract/convert from native data
This step takes the native EIS response data stream and converts it to XML in an IDocument
return IDocument
return IDocument

The Flow of Events

Figure 6-3 outlines the steps required to develop a Service Adapter.

Figure 6-3 Service Adapter Flow of Events



Step 1: Development Considerations

You will need to consider the items listed below before commencing with service adapter development. The Adapter Setup Worksheet will provide much of this information. See "Adapter Setup Worksheet."

- Determine the EIS and the appropriate service.

  You need to identify the EIS and the type of service required for this adapter; that is, based upon your knowledge of the EIS, you must identify the interface to the back-end functionality.

- Determine the expensive connection object.

  You need to determine the "expensive" connection object required to invoke a function within the EIS. The expensive connection object is a resource required to communicate with the EIS and requires the allocation of system resources; for example, a socket connection or DBMS connection. A valuable asset of J2EE Connector Architecture is that the application server provides pooling of these objects. Therefore, you must determine this object for your adapter, as it will need to be pooled by the application server.

- Determine the security needs.

  You need to consider and understand how to pass connection authentication across the connection request path. To do this, your adapter will need to implement a `connectionRequestInfo` class. The ADK provides the class `ConnectionRequestInfoMap` to map authorization information, such as username and password, to the connection to facilitate `ConnectionRequestInfo` implementation.

  The ADK conforms to the *J2EE Connector Architecture Specification*. For more information on connection architecture security, please refer to "Security" in that document. Go to http://java.sun.com/j2ee/connector/ to download the specification.

  The *J2EE Connector Architecture Specification* will download as a `.pdf` file.

- Determine transaction support.

  You need to identify which type of transaction demarcation support to implement with the adapter:

- Local transaction demarcation
- XA-compliant transaction demarcation

Note: For more information on transaction demarcation support, please see "Transaction Demarcation" or:

```
http://java.sun.com/j2ee/blueprints/
   transaction_management/platform/index.html
```

Step 2: Configuring the Development Environment

This step describes the processes you must complete to prepare your computer for adapter development.

Step 2a: Set Up the File Structure

Installing the Application Integration creates the file structure necessary not only to run an adapter, but also to use the ADK. The ADK files appear under <WLAI_HOME>\dev\, where <WLAI_HOME> is the directory where you installed the Application Integration distribution. You need to verify that, upon installation, the necessary directories and files appear in your <WLAI_HOME> directory. The file structure that follows under <WLAI_HOME>/ is described in Table 6-1:

Table 6-1 ADK File Structure

File Path/Filename	Description
dev	This directory contains the ADK.
dev/adk/src/war	All files under this directory should be included in the .war file for an adapter. This directory contains .jsp files, .html files, images, etc.

Table 6-1 ADK File Structure (Continued)

File Path/Filename	Description
dev/bin	This directory contains a file used by the build process to timestamp .jar files.
dev/dbms	This directory contains a sample J2EE-compliant adapter built with the ADK.
dev/email	This directory contains a sample J2EE-compliant adapter built with the ADK.
dev/doc	This directory should contain the user guide, release notes, and installation guide for an adapter.
dev/doc/api	This directory contains Javadoc for an adapter, the ADK, and related classes.
dev/project	This directory contains the primary make file (Makefile) for the ADK make process and its related MakeOptions files. You should consult MakeOptions.mak for instructions on using the make process.
dev/sample	This directory contains a sample adapter that you can use to start developing their own adapter.
dev/sample/docs	This directory should contain the user guide, release notes, and installation guide for a the sample adapter.
dev/sample/docs/api	This directory contains Javadoc for the sample adapter, the ADK, and related classes.
dev/sample/docs/api_internal	A directory for internal API documentation.
dev/sample/lib	This directory contains adapter related .jar, .rar, and .war files.
dev/sample/project	This directory contains the Apache Jakarta Ant build file build.xml. This file contains build information for compiling the source code, generating the .jar, .rar, and .war files, and generating Javadoc information. See "Step 2c: Setting Up the Build Process" for details on how to build the adapter.

Table 6-1 ADK File Structure (Continued)

File Path/Filename	Description
dev/sample/src	This directory contains all the source code for an adapter. It is up to you to decide to provide source code with your adapter.
dev/sample/src/BEA_WLS_SAMPLE_ADK.properties	This file contains messages used by the adapter for internationalization and localization.
dev/sample/src/BEA_WLS_SAMPLE_ADK.xml	This file provides a basic configuration file for the logging framework. You should use this file to develop their own adapter logging configuration file.
dev/sample/src/eventrouter/WEB-INF/ web.xml	This is the configuration file for the event router web application.
dev/sample/src/rar/META-INF/ra.xml	This file contains configuration information about a J2EE-compliant adapter. You should use this file as a guide on which parameters needed by the ADK's runtime framework.
dev/sample/src/rar/META-INF/weblogic-ra.xml	This file contains configuration information about a J2EE-compliant adapter that is specific to the WebLogic 6.0 J2EE engine. You should use this file as an example for setting up the weblogic-ra.xml file for their adapter. It is required for WebLogic 6.0
dev/sample/src/sample	This directory contains the source code for the adapter.
dev/sample/src/war	All files under this directory should be included in the web application archive (.war) file for an adapter. This directory contains .jsp files, .html files, images, etc. For more information on web applications for WebLogic 6.0, look here
dev/sample/src/war/WEB-INF/web.xml	The web application descriptor
dev/sample/src/war/WEB-INF/weblogic.xml	The weblogic.xml file contains WebLogic-specific attributes for a Web Application.

Modifying the Directory Structure

When you clone a development tree by using GenerateAdapterTemplate, the file paths and files under `dev/sample` are automatically cloned and updated to reflect the new development environment. The changes are reflected in the file `dev/<CLONE>/api/index.html` (where `<CLONE>` is the name of the new development directory). This file also contains code that you can copy and paste into the `config.xml` file and the `<DEPLOYMENT_HOME>/wlai.properties` files for the new adapter that will setup Application Integration to host the adapter.

Step 2b: Assign the Adapter Logical Name

Next, you need to assign the adapter's logical name. By convention, this name is comprised of the vendor name, the type of EIS connected to the adapter, and the version number of the EIS and is expressed as *vendor_EIS-type_EIS version.* For example:

```
BEA_WLS_SAMPLE_ADK
```

Step 2c: Setting Up the Build Process

The ADK employs a build process based upon Ant, a 100% pure Java-based build tool. For more information on Ant, please see "Ant-Based Build Process." For more information on using Ant, see http://jakarta.apache.org/ant/index.html.

The sample adapter shipped with the ADK (located in `<WLAI_HOME>\dev\bin\sample`) contains the file `build.xml`. This is the Ant build file for the sample adapter. It contains the tasks needed to build a J2EE-compliant adapter. Running the GenerateAdapterTemplate utility to clone a development tree for your adapter creates a `build.xml` file specifically for that adapter. This will free you from having to customize the sample `build.xml` and will ensure that the code is correct. For information on using the GenerateAdapterTemplate utility, see "Creating a Custom Development Environment."

build.xml Components

If you edit `build.xml` and review its components, you will better understand how this file works. This section describes the prominent elements of `build.xml`.

Note: The following examples are taken from the sample adapter, *not* a cloned version thereof.

1. The first line you encounter:

   ```
   <project name='BEA_WLS_SAMPLE_ADK' default='all' basedir='.'>
   ```

   sets the name attribute of the root project element.

2. Listing 6-1sets the archive file (`.jar`, `.war`, and `.rar`) names.

Listing 6-1 Setting Archive File Names

```
<property name='JAR_FILE' value='BEA_WLS_SAMPLE_ADK.jar'/>
<property name='RAR_FILE' value='BEA_WLS_SAMPLE_ADK.rar'/>
<property name='WLS_RAR_FILE' value='BEA_WLS_SAMPLE_ADK_
WLSRAR.rar'/><property name='WAR_FILE' value='BEA_WLS_SAMPLE_
ADK_Web.war'/>
```

3. Listing 6-2 shows a list of standard properties for the ADK. You shouldn't need to alter them.

Listing 6-2 Standard ADK Properties

```
<property name='PROJECT_DIR' value='.'/>
<property name='ROOT' value='${PROJECT_DIR}/../..'/>
<property name='SRC_DIR' value='${PROJECT_DIR}/../src'/>
<property name='LIB_DIR' value='${PROJECT_DIR}/../lib'/>
<property name='DOC_DIR' value='${PROJECT_DIR}/../docs/api'/>
<property name='RESOURCE_DIR' value='${ROOT}/../resources'/>
<property name='XALAN_JAR'
value='${RESOURCE_DIR}/xml/xalan.jar'/>
```

```
<property name='XERCES_JAR'
value='${RESOURCE_DIR}/xml/xerces_dp1.jar'/>

<property name='METAMATA_JAR'
value='${RESOURCE_DIR}/metamata/metamata.jar'/>

<property name='OROMATCHER_JAR'
value='${RESOURCE_DIR}/OROMatcher-1.1.0a/oromatcher.jar'/>

<property name='JAAS_JAR'
value='${RESOURCE_DIR}/jaas1.0/jaas.jar'/>

<property name='J2EECA_DIR' value='${RESOURCE_DIR}/j2eeca1.0'/>

<property name='J2EECA_JAR'
value='${J2EECA_DIR}/connector.jar'/>

<property name='JDK13_JAR'
value='${RESOURCE_DIR}/jre1.3/rt.jar'/>

<property name='J2EE_JAR'
value='${RESOURCE_DIR}/j2eesdk1.2/j2ee.jar'/>

<property name='LICENSE_JAR'
value='${RESOURCE_DIR}/license/license.jar'/>

<property name='LOG4J_JAR' value='${RESOURCE_DIR}/
log4j/log4j.jar'/>

<property name='WLAI_LIB_DIR' value='${ROOT}/../lib'/>

<property name='ADK' value='${WLAI_LIB_DIR}/adk.jar'/>

<property name='ADK_WEB' value='${WLAI_LIB_DIR}/adk-web.jar'/>

<property name='ADK_EVENTGENERATOR'
value='${WLAI_LIB_DIR}/adk-eventgenerator.jar'/>

<property name='BEA' value='${WLAI_LIB_DIR}/bea.jar'/>

<property name='LOGTOOLKIT'
value='${WLAI_LIB_DIR}/logtoolkit.jar'/>

<property name='WEBTOOLKIT'
value='${WLAI_LIB_DIR}/webtoolkit.jar'/>

<property name='WLAI_CLIENT'
value='${WLAI_LIB_DIR}/wlaiclient.jar'/>

<property name='WLAI_CLIENT_EJB'
value='${WLAI_LIB_DIR}/wlaiclient-ejb.jar'/>

<property name='WLAI_COMMON'
value='${WLAI_LIB_DIR}/wlai-common.jar'/>

<property name='WLAI_EJB'
value='${WLAI_LIB_DIR}/wlai-ejb.jar'/>
```

```
<property name='WLAI_EJB_CLIENT'
value='${WLAI_LIB_DIR}/wlai-ejb-client.jar'/>

<property name='WLAI_EVENTROUTER'
value='${WLAI_LIB_DIR}/wlai-eventrouter.jar'/>

<property name='WLAI_EVENTROUTER_CLIENT'
value='${WLAI_LIB_DIR}/wlai-eventrouter-client.jar'/>

<property name='WLAI_SERVLET_CLIENT'
value='${WLAI_LIB_DIR}/wlai-servlet-client.jar'/>

<property name='XMLTOOLKIT'
value='${WLAI_LIB_DIR}/xmltoolkit.jar'/>

<property name='XCCI' value='${WLAI_LIB_DIR}/xcci.jar'/>
```

To the list in Listing 6-2, you can add any additional `.jar` files and/or classes that are specific to your adapter.

4. Listing 6-3 sets up the `classpath` for compiling:

Listing 6-3 Sample for Setting the Classpath

```
<path id='CLASSPATH'>
   <pathelement location='${SRC_DIR}'/>
   <pathelement path='${ADK}:${ADK_WEB}:${BEA}:${LOGTOOLKIT}:
${WEBTOOLKIT}:${WLAI_COMMON}:${XCCI}:${XMLTOOLKIT}'/>
   <pathelement path='${J2EECA_JAR}:${JAAS_JAR}:${J2EE_JAR}'/>
   <pathelement path='${XERCES_JAR}:${XALAN_JAR}'/>
   <pathelement path='${LOG4J_JAR}'/>
</path>
```

To this information, you can add code that will produce the following:

- All the binaries and archives for the adapter:

  ```
  <target name='all' depends='rar,weblogic_rar,war'/>
  ```

- All the binaries and archives for the adapter, as well as the Javadoc:

```
<target name='release' depends='all,apidoc'/>
```

- A version_info file for inclusion into archives, as shown in Listing 6-4:

Listing 6-4 Sample version_info File

```
<target name='version_info'>
   <java classname='GenerateVersionInfo'>
     <classpath>
       <pathelement path='${ROOT}/bin:${XERCES_JAR}'/>
     </classpath>
   </java>
</target>
```

5. Listing 6-5 produce the `.jar` file for the adapter. This fileset element specifies what is included into the `.jar` file. Run-time aspects of the adapter are included in the main jar, while additional classes, such as the design-time GUI support classes, are included in the `.war` or other jar files.

Listing 6-5 Sample Setting .jar File Contents

```
<target name='jar' depends='packages,version_info'>
<delete file='${LIB_DIR}/${JAR_FILE}'/>
<jar jarfile='${LIB_DIR}/${JAR_FILE}'>
```

6. Listing 6-6 includes the "includes" list from the adapter's source directory. For the adapter described in these code samples, all the classes in the `sample/cci` and `sample/spi` packages are included, as well as the logging configuration file and message bundles.

Listing 6-6 Sample Code for Including the "Includes" List

```
<fileset dir='${SRC_DIR}'
```

```
includes='sample/cci/*.class,sample/ spi/*.class,*.xml,*.
properties'/>
```

7. Listing 6-7 includes version information about the `.jar` file:

Listing 6-7 Setting .jar File Version Information

```
<fileset dir='${PROJECT_DIR}'
includes='version_info.xml'/>
</jar>
</target>
```

8. Listing 6-8 produces the J2EE adapter archive (`.rar`) file. The `.rar` file should contain all classes and JARs that the adapter needs. This `.rar` can be deployed into any J2EE-compliant application server that the adapter depends upon. This example includes the following targets:
   - The adapter's main `.jar` file.
   - Version information for this `.rar` file.
   - The `deployment descriptor` for the adapter.
   - The required ADK `.jar` files used by the adapter, the ADK.
   - The Apache log4j `.jar` used by the logging toolkit.
   - The XML tools needed by the adapter and the ADK.

Listing 6-8 Sample Code for Creating the Connection Architecture .rar File

```
<target name='rar' depends='jar'>
    <delete file='${LIB_DIR}/${RAR_FILE}'/>
    <mkdir dir='${LIB_DIR}'/>
    <jar jarfile='${LIB_DIR}/${RAR_FILE}'>
```

```
      <!-- Include the adapter's main jar file -->
      <fileset dir='${LIB_DIR}' includes='${JAR_FILE}'/>

      <!-- Include version information for this RAR file -->
      <fileset dir='${PROJECT_DIR}'includes= 'version_info.xml'/>

      <!-- Include the deployment descriptor for the resource adapter -->
      <fileset dir='${SRC_DIR}/rar' includes='META-INF/ra.xml'/>

      <!-- Include the required WLIF jars used by the adapter, ADK etc. -->
      <fileset dir='${WLAI_LIB_DIR}' includes='adk.jar,bea.jar,logtoolkit.
jar,wlai-common.jar,wlai-ejb-client.jar,xcci.jar,xmltoolkit.jar'/>

      <!-- Include the Apache LOG4J jar used by the logging toolkit -->
      <fileset dir='${RESOURCE_DIR}/log4j' includes='log4j.jar'/>
    </jar>
  </target>
```

9. Listing 6-9 produces the `.rar` file specific for the WebLogic 6.0 platform. This file contains the `weblogic-ra.xml` file for the adapter. Otherwise, the fileset is the same for the standard `.rar`.

Listing 6-9 Creating the WebLogic 6.0-Specific .rar File

```
<target name='weblogic_rar' depends='jar'>
<delete file='${LIB_DIR}/${WLS_RAR_FILE}'/>
  <jar jarfile='${LIB_DIR}/${WLS_RAR_FILE}'>
     <fileset dir='${LIB_DIR}' includes='${JAR_FILE}'/>
     <fileset dir='${PROJECT_DIR}' includes='version_info. xml'/>
```

```
     <fileset dir='${SRC_DIR}/rar' includes='META-INF/
ra.xml,META-INF/weblogic-ra.xml'/>

     <fileset dir='${WLAI_LIB_DIR}' includes='adk.jar,bea.
jar,logtoolkit.jar,wlai-common.jar,wlai-ejb-client.jar,xcci.jar,
xmltoolkit.jar'/>

     <fileset dir='${RESOURCE_DIR}/log4j' includes='log4j. jar'/>

     <fileset dir='${RESOURCE_DIR}/xml' includes='xerces_dp1. jar'/>

     <fileset dir='${RESOURCE_DIR}/xml' includes='xalan.jar'/>

  </jar>

</target>
```

10. Listing 6-10 produces the J2EE web application archive (`.war`) file. It also includes code that cleans up the existing environment:

Listing 6-10 Sample Code Producing the `.war` File

```
   <target name='war' depends='jar'>
      <!-- Clean-up existing environment -->
      <delete file='${LIB_DIR}/${WAR_FILE}'/>
      <delete dir='${SRC_DIR}/war/WEB-INF/lib'/>
      <delete dir='${SRC_DIR}/war/WEB-INF/classes'/>
      <war warfile='${LIB_DIR}/${WAR_FILE}'
  webxml='${SRC_DIR}/war/WEB-INF/web.xml'>
       <fileset dir='${PROJECT_DIR}' includes='version_info.xml'/>
        <!--
        IMPORTANT! Exclude the WEB-INF/web.xml file from the WAR
        as it already gets included via the webxml attribute above
        -->
       <fileset dir='${SRC_DIR}/war' excludes='WEB-INF/web.xml'/>
       <!--
        IMPORTANT! Include the ADK design time framework into the
```

```
      adapter's design time Web application.
      -->
       <fileset dir='${ROOT}/adk/src/war'/>
       <!-- Include classes from the adapter that support the
design time UI -->
       <classes dir='${SRC_DIR}' includes='sample/web/*.class'/>
       <!--
       Include all JARs required by the Web application under the
      WEB-INF/lib directory of the WAR file
       -->
       <lib dir='${LIB_DIR}' includes='${JAR_FILE}'/>
       <lib dir='${WLAI_LIB_DIR}'
includes='adk.jar,adk-web.jar,bea.jar,logtoolkit.jar,webtoolkit
.jar,wlai-common.jar,wlai-ejb-client.jar,xcci.jar,xmltoolkit.ja
r'/>
       <lib dir='${RESOURCE_DIR}/log4j' includes='log4j.jar'/>
       <lib dir='${RESOURCE_DIR}/OROMatcher-1.1.0a'
includes='oromatcher.jar'/>
     </war>
```

11. Listing 6-11 includes all `.jar` files required by the web application under the `WEB-INF/lib` directory of the `.war` file.

Listing 6-11 Sample Code to Include JARs Required by Web Application

```
<lib dir='${LIB_DIR}' includes='${JAR_FILE}'/>
     <lib dir='${WLAI_LIB_DIR}'
includes='adk.jar,adk-web.jar,bea.jar,logtoolkit.jar,webtoolkit
.jar,wlai-common.jar,wlai-ejb-client.jar,xcci.jar,xmltoolkit.ja
r'/>
     <lib dir='${RESOURCE_DIR}/log4j' includes='log4j.jar'/>
     <lib dir='${RESOURCE_DIR}/OROMatcher-1.1.0a'
includes='oromatcher.jar'/>
     <lib dir='${RESOURCE_DIR}/xml' includes='xerces_dp1.jar'/>
```

```
      <lib dir='${RESOURCE_DIR}/xml' includes='xalan.jar'/>
    </war>
  </target>
```

12. Listing 6-12 compiles all the Java source files for this project:

Listing 6-12 Sample Code for Compiling Java Source

```
<target name='packages'>
   <javac srcdir='${SRC_DIR}'>
     <classpath refid='CLASSPATH'/>
   </javac>
</target>
```

13. Listing 6-13 generates the Javadoc.

Listing 6-13 Sample Code for Generating Javadocs

```
<target name='apidoc'>
<mkdir dir='${DOC_DIR}'/>
   <javadoc sourcepath='${SRC_DIR}'
     destdir='${DOC_DIR}'
     packagenames='sample.*'
     author='true'
     version='true'
     use='true'
     windowtitle='WebLogic Sample Adapter API Documentation'
     doctitle='WebLogic Sample Adapter API Documentation'
     header='WebLogic Sample Adapter'
```

```
     bottom='Built using the WebLogic Adapter Development Kit
(ADK)'>
     <classpath refid='CLASSPATH'/>
   </javadoc>
</target>
```

14. Listing 6-14 shows the targets that clean the files created by their counterparts:

Listing 6-14 Sample Code for Including Clean-Up Code

```
<target name='clean_release' depends='clean_all,clean_apidoc'/>
  <target name='clean_all'
depends='clean_rar,clean_weblogic_rar, clean_war'/>
  <target name='clean_rar' depends='clean_jar'>
     <delete file='${LIB_DIR}/${RAR_FILE}'/>
  </target>
  <target name='clean_weblogic_rar' depends='clean_jar'>
     <delete file='${LIB_DIR}/${WLS_RAR_FILE}'/>
  </target>
  <target name='clean_war' depends='clean_jar'>
     <delete file='${LIB_DIR}/${WAR_FILE}'/>
  </target>
  <target name='clean_jar' depends='clean_packages,
clean_version_info'>
     <delete file='${LIB_DIR}/${JAR_FILE}'/>
  </target>
  <target name='clean_version_info'>
     <delete file='${PROJECT_DIR}/version_info.xml'/>
  </target>
  <target name='clean_packages'>
     <delete>
```

```
        <fileset dir='${SRC_DIR}' includes='**/*.class'/>
      </delete>
    </target>
    <target name='clean_apidoc'>
      <delete dir='${DOC_DIR}'/>
    </target>
  </project>
```

Step 2d: Create the Message Bundle

Any message destined for the end-user should be placed in a message bundle. The message bundle is simply a `.properties` text file that contains key=value pairs that allow you to internationalize messages. When a locale and national language are specified at runtime, the contents of the message is interpreted, based upon the key=value pair, and the message is presented to the user in the correct language for his or her locale.

For instructions on creating a message bundle, please refer to the JavaSoft tutorial on internationalization at:

`http://java.sun.com/docs/books/tutorial/i18n/index.html`

Step 3: Implementing the SPI

The Service Provider Interface (SPI) contains the objects that provide and manage connectivity to the EIS, establish transaction demarcation, and provide a framework for service invocation. All J2EE-compliant adapters must provide an implementation for these interfaces in the `javax.resource.spi` package.

How to Use this Section

"Step 3: Implementing the SPI" contains descriptions of the interfaces you can use to implement the SPI. A minimum of three interfaces are necessary to complete the task (see "Basic SPI Implementation"). Each of these are described in detail, followed by a discussion of how they were extended in the sample adapter included with the ADK.

Following the three required interfaces, the additional interfaces are described in detail, including information regarding why you might use them and what benefit they are to an adapter.

Basic SPI Implementation

To implement the SPI for your adapter, you need to extend *at least* these three interfaces:

- `ManagedConnectionFactory`, which supports connection pooling by providing methods for matching and creating a ManagedConnection instance.
- `ManagedConnection`, which represents a physical connection to the underlying EIS.
- `ManagedConnectionMetaData`, which provides information about the underlying EIS instance associated with a `ManagedConnection` instance.

Ideally, you will implement these interfaces in the order specified above.

In addition to these three interfaces, you can implement any of the other interfaces described in this step, as your adapter needs dictate.

ManagedConnectionFactory

`javax.resource.spi.ManagedConnectionFactory`

`ManagedConnectionFactory` instance is a factory of both ManagedConnection and EIS-specific connection factory instances. This interface supports connection pooling by providing methods for matching and creating a `ManagedConnection` instance.

Transaction Demarcation

A critical component of the ManagedConnectionFactory is transaction demarcation. You will need to determine which statements in your program are included in a single transaction. J2EE defines a transaction management contract between an application server and an adapter (and its underlying resource manager). The transaction management contract has two parts, depending of the type of transaction:

- XA-compliant Transaction
- Local Transaction

XA-compliant Transaction

A **`javax.transaction.xa.XAResource`**-based contract occurs between a transaction manager and a resource manager in a distributed transaction processing (DTP) environment. A JDBC driver or a JMS provider implements this interface to support association between a global transaction and a database or message service connection.

The XAResource interface can be supported by any transactional resource that is intended for use by application programs in an environment where transactions are controlled by an external transaction manager; for example a database management system where an application accesses data through multiple database connections. Each database connection is enlisted with the transaction manager as a transactional resource. The transaction manager obtains an XAResource for each connection participating in a global transaction. The transaction manager uses the `start()` method to associate the global transaction with the resource; it uses the `end()` method to disassociate the transaction from the resource. The resource manager associates the global transaction to all work performed on its data between the `start()` and `end()` method invocation.

At transaction commit time, the resource managers are informed by the transaction manager to prepare, commit, or rollback a transaction according to the two-phase commit protocol.

Local Transaction

A **local transaction management contract** occurs when an adapter implements the `javax.resource.spi.LocalTransaction` interface to provide support for local transactions that are performed on the underlying resource manager. These contracts

enable an application server to provide the infrastructure and runtime environment for transaction management. Application components rely on this transaction infrastructure to support their component-level transaction model.

For more information on transaction demarcation support, please refer to:

http://java.sun.com/j2ee/blueprints/transaction_management/ platform/index.html

ADK Implementations

The ADK provides an abstract foundation for an adapter, the AbstractManagedConnectionFactory. This foundation provides the following feature:

- Provides basic support for internationalization/localization of exception and log messages for an adapter.
- Provides hooks into the log toolkit.
- Provides getter and setter methods for standard connection properties (username, password, server, connectionURL, port)
- Provides access to adapter metadata gathered from a `java.util.ResourceBundle` for an adapter.
- Allows adapter providers to plug in license checking into the initialization process for the factory. If the license verification fails, then client applications cannot get a connection to the underlying EIS, thus making the adapter useless.
- Provides state verification checking to support JavaBeans-style post-constructor initialization.

There are several key methods that you must supply implementations for. The following paragraphs describe these methods.

createConnectionFactory()

`createConnectionFactory()`, shown in Listing 6-15, is responsible for constructing the factory for application-level connection handles for the adapter. In other words, clients of your adapter will use the object returned by this method to obtain a connection handle to the EIS.

If the adapter supports a CCI interface, it is recommended that you return an instance of `com.bea.adapter.cci.ConnectionFactoryImpl` or an extension of this class. The key to implementing this method correctly is to propagate the ConnectionManager, LogContext, and ResourceAdapterMetaData into the client API.

Listing 6-15 createConnectionFactory() Example

```
protected Object
    createConnectionFactory(ConnectionManager connectionManager,
                            String strAdapterName,
                            String strAdapterDescription,
                            String strVendorName)
    throws ResourceException
```

createManagedConnection()

`createManagedConnection()`, shown in Listing 6-16, is responsible for constructing a ManagedConnection instance for your adapter. The ManagedConnection instance encapsulates the expensive resources needed to communicate with the EIS. This method is called by the ConnectionManager when it determines a new ManagedConnection is required to satisfy a client's request. A common design pattern with adapters is to open the resources needed to communicate with the EIS in this method and then pass the resources into a new ManagedConnection instance.

Listing 6-16 createManagedConnection() Example

```
public ManagedConnection
   createManagedConnection(Subject subject, ConnectionRequestInfo
      info)
     throws ResourceException
```

checkState()

`checkState()` gets called by the AbstractManagedConnectionFactory before it attempts to perform any of its factory responsibilities. Use this method to verify that all members that need to be initialized before the ManagedConnectionFactory can perform its SPI responsibilities have been initialized correctly. Implement this method as shown here:

```
protected boolean checkState()
```

equals()

`equals()` tests the object argument for equality. It is important to implement this method correctly as it is used by the ConnectionManager for managing the connection pools. This method should include all important members in its equality comparison. Implement this method as shown here:

```
public boolean equals(Object obj)
```

hashCode()

`hashCode()` provides a hash code for the factory. It is also used by the ConnectionManager for managing the connection pools. Consequently, this method should generate a hashCode based upon properties that determine the uniqueness of the object. Implement this method as shown here:

```
public int hashCode()
```

matchManagedConnections()

Lastly, the ManagedConnectionFactory must supply an implementation of the `matchManagedConnections()` method. The AbstractManagedConnectionFactory provides an implementation of the `matchManagedConnections()` method that relies upon the compareCredentials method on AbstractManagedConnection.

In order to provide logic that will match managed connections, you will need to override `AbstractManagedConnection`'s `compareCredentials()` method. This method is invoked when the ManagedConnectionFactory attempts to match a connection with a connection request for the ConnectionManager.

Currently, AbstractManagedConnectionFactory's implementation extracts a `PasswordCredential` from the supplied `Subject/ConnectionRequestInfo` parameters. If both parameters are null, this method returns true because it has already been established that the ManagedConnectionFactory for this instance is correct. Listing 6-17 shows this implementation:

Listing 6-17 compareCredentials() Implementation

```
public boolean compareCredentials(Subject subject,
                          ConnectionRequestInfo info)
          throws ResourceException
     {
       ILogger logger = getLogger();
```

Next, you need to extract a PasswordCredential from either the JAAS Subject or the SPI ConnectionRequestInfo using the ADK's `ManagedConnectionFactory`. An example is shown in Listing 6-18:

Listing 6-18 Extracting a PasswordCredential

```
PasswordCredential pc = getFactory().
getPasswordCredential(subject, info);
     if (pc == null)
     {
       logger.debug(this.toString() + ": compareCredentials
```

In the example shown in Listing 6-18, JAAS Subject and ConnectionRequestInfo are null, which assumes a match. This method will not get invoked unless it has already been established that the factory for this instance is correct. Consequently, if the Subject and ConnectionRequestInfo are both null, then the credentials match by default; therefore, the result of pinging this connection determines the outcome of the comparison. Listing 6-19 shows how to programmatically ping the connection.

Listing 6-19 Pinging a Connection

```
      return ping();
    }
    boolean bUserNameMatch = true;
    String strPcUserName = pc.getUserName();
    if (m_strUserName != null)
    {
      logger.debug(this.toString() + ": compareCredentials >>>
comparing my username ["+m_strUserName+"] with client username
["+strPcUserName+"]");
```

Next, you need to see if the user supplied in either the Subject or the ConnectionRequestInfo is the same as our user. We do not support re-authentication in this adapter, so if they do not match, this instance cannot satisfy the request. The following line of code does that:

```
bUserNameMatch = m_strUserName.equals(strPcUserName);
```

If usernames match, ping the connection to determine if this is still a good connection. Otherwise, there is no match and no reason to ping. The following line of code does that:

```
return bUserNameMatch ? ping() : false;
```

Explanation of the Implementation

Under a managed scenario, the application server invokes the `matchManagedConnections()` method on the ManagedConnectionFactory for an adapter. The specification does not indicate how the application server determines which ManagedConnectionFactory to use to satisfy a connection request. The ADK's AbstractManagedConnectionFactory implements `matchManagedConnections()`. The first step in this implementation is to compare "this" (that is, the ManagedConnectionFactory instance on which the ConnectionManager invoked matchManagedConnections) to the ManagedConnectionFactory on each ManagedConnection in the set supplied by the application server. For each

ManagedConnection in the set that has the same ManagedConnectionFactory, the implementation invokes the `compareCredentials()` method. This method allows each ManagedConnection object to determine if it can satisfy the request.

`matchManagedConnections()` gets called by the ConnectionManager (as shown in Listing 6-20) to try to find a valid connection in the pool it is managing. If this method returns null, then the ConnectionManager will allocate a new connection to the EIS via a call to `createManagedConnection()`.

Listing 6-20 matchManagedConnections() Method Implementation

```
public ManagedConnection
matchManagedConnections(Set connectionSet,
                        Subject subject,
                        ConnectionRequestInfo info)
      throws ResourceException
```

This class uses the following approach to matching a connection:

1. It iterates over the <code>connectionSet</code>, for each object in the set (until a match is found) it determines whether to not it's an AbstractManagedConnection.
2. If it is, then this is compared to the ManagedConnectionFactory for the AbstractManagedConnection from the set.
3. If the factories are equal, then the `compareCredentials()` method is invoked on the AbstractManagedConnection.
4. If this method returns true, then the instance is returned.

AbstractManagedConnectionFactory Properties Required at Deployment

To use the base implementation of AbstractManagedConnectionFactory, you need to provide the following properties at deployment time:

Table 6-2 AbstractManagedConnectionFactory Properties

Property Name	Property Type	Applicable Values	Description	Default
`LogLevel`	`java.lang.String`	`ERROR, WARN, INFO, DEBUG`	Logs verbosity level.	`WARN`
`LanguageCode`	`java.lang.String`	valid ISO Language Code, see `http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt`	Determines the desired locale for log messages	`en`
`CountryCode`	`java.lang.String`	valid ISO Country Code, see `http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html`	Determines the desired locale for log messages	`US`
`MessageBundleBase`	`java.lang.String`	Any valid Java Class name or file name	Determines the message bundle for log messages	none, required
`LogConfigFile`	`java.lang.String`	Any valid file name	Configures the LOG4J system	none, required
`RootLogContext`	`java.lang.String`	Any valid Java String	Categorizes log messages from this connection factory.	none, required
`AdditionalLogContext`	`java.lang.String`	Any valid Java String	Adds additional information to uniquely identify messages from this factory.	none, optional

Other Key ManagedConnectionFactory Features in the ADK

In the ADK sample adapter, the class `sample.spi.ManagedConnectionFactoryImpl` is provided. This class extends `AbstractManagedConnectionFactory`. Use this class as an example of how to extend the ADK's base class.

For the complete sample adapter ManagedConnectionFactory implementation code listing, see:

`<WLAI_HOME>\dev\sample\src\sample\spi\ManagedConnectionFactoryImpl.java`

ManagedConnection

`javax.resource.spi.ManagedConnection`

The ManagedConnection object is responsible for encapsulating all expensive resources needed to establish connectivity to the EIS. A ManagedConnection instance represents a physical connection to the underlying EIS. ManagedConnection objects are pooled by the application server in a managed environment.

ADK Implementation

The ADK provides an abstract implementation of this interface. The base class provides logic for managing connection event listeners and multiple application-level connection handles per ManagedConnection instance.

When implementing the ManagedConnection interface, you need to determine the transaction demarcation support provided by the underlying EIS. For more information on transaction demarcation, see "Transaction Demarcation."

The ADK provides AbstractManagedConnection, an abstract implementation for the `javax.resource.spi.ManagedConnection` interface that:

- Provides access to the ADK logging framework.
- Manages a collection connection event listeners.
- Provides convenience methods for notifying all connection event listeners of connection related events.

- Simplifies clean-up and destruction of a ManagedConnection instance.

The sample adapter that comes with the ADK includes ManagedConnectionImpl, which extends AbstractManagedConnection. For the complete sample adapter `ManagedConnection` implementation code listing, see:

```
<WLAI_HOME>\dev\sample\src\sample\spi\ManagedConnectionFactory
Impl.java
```

ManagedConnectionMetaData

`javax.resource.spi.ManagedConnectionMetaData`

The ManagedConnectionMetaData interface provides information about the underlying EIS instance associated with a ManagedConnection instance. An application server uses this information to get runtime information about a connected EIS instance.

ADK Implementation

The ADK provides AbstractManagedConnectionMetaData, an abstract implementation of the `javax.resource.spi.ManagedConnectionMetaData` and `javax.resource.cci.ConnectionMetaData` interfaces that:

- Simplifies exception handling.
- Provides access to an AbstractManagedConnection instance.
- Allows you to focus on implementing EIS-specific logic.
- Prevents you from having a separate metadata class for the CCI and SPI implementations.

The sample adapter that comes with the ADK includes ManagedConnectionMetaDataImpl, which extends AbstractManagedConnectionMetaData. For the complete code listing, see:

```
<WLAI_HOME>\dev\sample\src\sample\spi\ManagedConnectionMetaDataIm
   pl. java
```

ConnectionEventListener

javax.resource.spi.ConnectionEventListener

The ConnectionEventListener interface provides an event callback mechanism that enables an application server to receive notifications from a ManagedConnection instance.

ADK Implementation

The ADK provides two concrete implementations of this interface:

- `com.bea.adapter.spi.ConnectionEventLogger`, which logs connection-related events to the adapter's log by using the ADK logging framework.
- `com.bea.adapter.spi.NonManagedConnectionEventListener`, which destroys `javax.resource.spi.ManagedConnection` instances when the adapter is running in a non-managed environment. This implementation:
  - Logs connection-related events using the ADK logging framework.
  - Destroys ManagedConnection instances when a connection related error occurs.

In most cases, the implementations provided by the ADK are sufficient; you should not need to provide your own implementation of this interface.

ConnectionManager

javax.resource.spi.ConnectionManager

The ConnectionManager interface provides a hook for the adapter to pass a connection request to the application server.

ADK Implementation

The ADK provides a concrete implementation of this interface, `com.bea.adapter.spi.NonManagedConnectionManager`. This implementation provides a basic connection manager for adapters running in a non-managed environment. In a managed environment, this interface is provided by the application server. In most cases, you can use the implementation provided by the ADK.

NonManagedConnectionManager is a concrete implementation of the `javax.resource.spi.ConnectionManager` interface. It serves as the ConnectionManager in the non-managed scenario for an adapter; it does not provide any connection pooling or any other quality of service.

ConnectionRequestInfo

`javax.resource.spi.ConnectionRequestInfo`

The ConnectionRequestInfo interface enables an adapter to pass its own request specific data structure across the connection request flow. An adapter extends the empty interface to support its own data structures for a connection request.

ADK Implementation

The ADK provides a concrete implementation of this interface called ConnectionRequestInfoMap. This is a concrete implementation of the `javax.resource.spi.ConnectionRequestInfo` interface and provides a `java.util.Map` interface to such connection request information as username and password.

LocalTransaction

`javax.resource.spi.LocalTransaction`

The LocalTransaction interface provides support for transactions that are managed internal to an EIS resource manager, and do not require an external transaction manager.

ADK Implementation

The ADK provides an abstract implementation of this interface called AbstractLocalTransaction. This implementation allows you to focus on implementing the EIS-specific aspects of a LocalTransaction. This implementation:

- Simplifies exception handling.
- Allows adapter providers to focus on implementing EIS-specific transaction logic.
- Prevents you from having a separate metadata class for the CCI and SPI implementations.

Step 4: Implementing the CCI

The client interface allows a J2EE-compliant application to connect to and access back-end systems. The client interface manages the flow of data between the client application and the back-end system and does not have any visibility into what either the container or the application server are doing with the adapter. The client interface specifies the format of the request and response records for a given interaction with the EIS.

First, you must determine if your adapter must support the J2EE-compliant Common Client Interface (CCI). Although not a requirement in the 1.0 J2EE specification, it is likely to be a requirement in the next version. Consequently, the ADK focuses on helping you implement a CCI interface for your adapter.

How to Use this Section

"Step 4: Implementing the CCI" describes some of the interfaces you can use to implement the CCI. A minimum of two interfaces are necessary to complete the task (see "Basic CCI Implementation"). Each of these is described in detail, followed by a discussion of how they were extended in the sample adapter included with the ADK.

Following the two required interfaces, the additional interfaces are described in detail, including information regarding why you might use them and what benefit they provide to an adapter.

Basic CCI Implementation

To implement the CCI for your adapter, you need to extend *at least* these two interfaces:

- Connection, which represents an application-level handle that is used by a client to access the underlying physical connection.
- Interaction, which enables a component to execute EIS functions.

Ideally, you will implement these interfaces in the order specified above.

In addition to these interfaces, you can implement any of the other interfaces described in this step, as your adapter needs dictate. These interfaces are:

- ConnectionFactory
- ConnectionMetaData
- ConnectionSpec
- InteractionSpec
- LocalTransaction
- Record
- ResourceAdapterMetaData

Connection

`javax.resource.cci.Connection`

A Connection represents an application-level handle that is used by a client to access the underlying physical connection. The actual physical connection associated with a Connection instance is represented by a ManagedConnection instance.

A client gets a Connection instance by using the getConnection method on a ConnectionFactory instance. A connection can be associated with zero or more Interaction instances.

ADK Implementation

The ADK provides an abstract implementation of this interface called AbstractConnection. This interface provides the following functionality:

- Access to the ADK logging framework.
- Access to an AbstractManagedConnection instance
- State management and assertion checking.

You will need to extend this class by providing an implementation for:

```
public Interaction createInteraction()
        throws ResourceException
```

This method creates an interaction associated with this connection. An interaction enables an application to execute EIS functions. This method:

- Returns: Interaction instance.
- Throws: ResourceException - Exception if the create operation fails.

Interaction

`javax.resource.cci.Interaction`

The `javax.resource.cci.Interaction` enables a component to execute EIS functions. An Interaction instance supports the following ways of interacting with an EIS instance:

- An `execute()` method that takes an input Record, output Record, and an InteractionSpec. This method executes the EIS function represented by the InteractionSpec and updates the output Record
- An `execute()` method that takes an input Record and an InteractionSpec. This method implementation executes the EIS function represented by the InteractionSpec and produces the output Record as a return value.

An interaction instance is created from a connection and is required to maintain its association with the Connection instance. The `close()` method releases all resources maintained by the adapter for the interaction. The close of an interaction instance should not close the associated connection instance.

ADK Implementation

The ADK provides an implementation of this interface called AbstractInteraction. This interface:

- Provides access to the ADK logging framework.
- Manages warnings.

You must supply a concrete extension to AbstractInteraction that implements `execute()`. Use at least one of the following versions of `execute()`:

execute() Version 1

The `execute()` method declared in Listing 6-21 shows an interaction represented by the InteractionSpec. This form of invocation takes an input record and updates the output record.

This method:

- Returns true if execution of the EIS function has been successful and output Record has been updated; otherwise it returns false.
- Throws ResourceException - Exception if execute operation fails

Listing 6-21 `execute()` Version 1 Code Example

```
public boolean execute(InteractionSpec ispec,
                       Record input,
                       Record output)
               throws ResourceException
```

The parameters for `execute()` version 1 are:

Table 6-3 `execute()` Version 1 Parameters

Parameters	Description
ispec	InteractionSpec representing a target EIS data/function module
input	Input Record
output	Output Record

execute() Version 2

The `execute()` method declared in Listing 6-22 also executes an interaction represented by the InteractionSpec. This form of invocation takes an input Record and returns an output record if the execution of the Interaction has been successful.

This method:

- Returns an output record if execution of the EIS function has been successful; otherwise it throws an exception.
- Throws ResourceException - Exception if execute operation fails.

If an exception occurs, this method will notify its connection, which will take the appropriate action, including closing itself.

Listing 6-22 execute() Version 2 Code Example

```
public Record execute(InteractionSpec ispec,
                      Record input)
              throws ResourceException
```

The parameters for `execute()` version 2 are:

Table 6-4 execute() Version 2 Parameters

Parameter	Description
ispec	InteractionSpec representing a target EIS data/function module
input	Input Record

Using XCCI to Implement the CCI

XCCI (XML-CCI) It is a dialect of CCI that uses XML-based record formats to represent data. It provides the tools and framework for supporting this record format. There are two primary components of XCCI: Services and DocumentRecords.

A service represents functionality available in an EIS and is comprised of four components:

- Unique Business Name

  Every service has a unique business name that indicates its role in an integration solution. For example, in an integration solution involving a Customer Relationship Management (CRM) system, you may have a service named "CreateNewCustomer". It is important to understand that the service name should reflect the business purpose of the service; it is an abstraction from the name of the function(s) your service invokes in the EIS

- Request Document Definition

  The request document definition describes the input requirements for a service. The `com.bea.document.IDocumentDefinition` interface embodies all the metadata about a document type. It includes the document schema (structure and usage), and the root element name for all documents of this type. The root element name is needed because an XML schema can define more than one possible root element.

- Response Document Definition

  The response document definition describes the output for a service.

- Additional Metadata

A service is a higher-order component in an integration solution that hides most of the complexity involved in executing functionality in an EIS. In other words, a service does not expose many of the details required to interact with the EIS in its public interface. This implies that some of the information required to invoke a function in an EIS is not provided by the client in the request. Consequently, most services need to store additional metadata. In Application Integration, this additional metadata is encapsulated by an adapter's `javax.resource.cci.InteractionSpec` implementation class.

DocumentRecord

`com.bea.connector.DocumentRecord`

At runtime, the XCCI layer expects DocumentRecord objects as input to a service and returns DocumentRecord objects as output from a service. DocumentRecord implements the `javax.resource.cci.Record` and the `com.bea.document.IDocument` interfaces. See "Record" for a description of that interface. IDocument, which facilitates XML input and output from the CCI layer in an adapter, is described in the following section.

IDocument

`com.bea.document.IDocument`

An IDocument is a higher-order wrapper around the W3C Document Object Model (DOM). The primary value-add of the IDocument interface is it provides an XPath interface to elements in an XML document. In other words, IDocument objects are queryable and updatable using XPath strings. For example, The XML document shown in Listing 6-23 describes a person named "Bob" and some of the details about "Bob."

Listing 6-23 XML Example

```
<Person name="Bob">
   <Home squareFeet="2000"/>
   <Family>
     <Child name="Jimmy">
       <Stats sex="male" hair="brown" eyes="blue"/>
```

```
      </Child>
      <Child name="Susie">
        <Stats sex="female" hair="blonde" eyes="brown"/>
      </Child>
    </Family>
</Person>
```

By using IDocument, you can retrieve Jimmy's hair color using the code shown in Listing 6-24:

Listing 6-24 IDocument Data Retrieval Code Sample

```
System.out.println("Jimmy's hair color: " +
person.getStringFrom("//Person[@name=\"Bob\"]/Family/Child[@name=
\"Jimmy\"]/Stats/@hair");
```

On the other hand, if you used DOM, you would need to enter the code shown in Listing 6-25:

Listing 6-25 DOM Data Retrieval Code Sample

```
String strJimmysHairColor = null;
org.w3c.dom.Element root = doc.getDocumentElement();
if (root.getTagName().equals("Person") &&
    root.getAttribute("name").equals("Bob") {
  org.w3c.dom.NodeList list = root.getElementsByTagName("Family");
  if (list.getLength() > 0) {
    org.w3c.dom.Element family = (org.w3c.dom.Element)list.item(0);
    org.w3c.dom.NodeList childList =
family.getElementsByTagName("Child");
```

```
    for (int i=0; i < childList.getLength(); i++) {
      org.w3c.dom.Element child = childList.item(i);
      if (child.getAttribute("name").equals("Jimmy")) {
        org.w3c.dom.NodeList statsList =
child.getElementsByTagName("Stats");
        if (statsList.getLength() > 0) {
          org.w3c.dom.Element stats = statsList.item(0);
          strJimmysHairColor = stats.getAttribute("hair");
        }
      }
    }
  }
}
```

As you can see, by using IDocument, you can simplify your code.

ADK-Supplied XCCI Classes

The ADK provides several classes that will help you implement XCCI for your adapters. This section describes those classes.

AbstractDocumentRecordInteraction

`com.bea.adapter.cci.AbstractDocumentRecordInteraction`

This class extends the ADK's abstract base Interaction, `com.bea.adapter.cci.AbstractInteraction`. The purpose of this class is to provide convenience methods for manipulating DocumentRecords and to reduce the amount of error handling the you need to implement. Specifically, this class declares:

```
protected abstract boolean execute(InteractionSpec ixSpec,
DocumentRecord inputDoc, DocumentRecord outputDoc) throws
ResourceException
```

and

```
protected abstract DocumentRecord execute(InteractionSpec ixSpec,
DocumentRecord inputDoc) throws ResourceException
```

These methods will not be invoked on the concrete implementation until the parameters have been verified that they are DocumentRecord objects.

DocumentDefinitionRecord

`com.bea.adapter.cci.DocumentDefinitionRecord`

This class allows the adapter to return an IDocumentDefinition from its DocumentRecordInteraction implementation. This class is useful for satisfying design-time requests to create the request and/or response document definitions for a service.

ServiceInteractionSpecImpl

`com.bea.adapter.cci.ServiceInteractionSpecImpl`

This class allows you to save the request document definition and response document definition for a service into the InteractionSpec provided to the execute method at runtime. This is useful when the Interaction for an adapter needs access to the XML schemas for a service at runtime.

XCCI Design Pattern

A common design pattern that emerges when using the XCCI approach is to support the definition of services in the Interaction implementation. In other words, the `javax.resource.cci.Interaction` implementation for an adapter allows a client program to retrieve metadata from the underlying EIS in order to define an Application Integration service. Specifically, this means that the interaction must be able to generate the request and response XML schemas and additional metadata for a service. Additionally, the Interaction could also allow a client program to browse a catalog of functions provided by the EIS. This approach facilitates a thin client architecture for your adapter.

The ADK provides the `com.bea.adapter.cci.DesignTimeInteractionSpecImpl` class to help you implement this design pattern. The `sample.cci.InteractionImpl` class demonstrates how to implement this design pattern using the DesignTimeInteractionSpecImpl class.

ConnectionFactory

`javax.resource.cci.ConnectionFactory`

`ConnectionFactory` provides an interface for getting connection to an EIS instance. An implementation of ConnectionFactory interface is provided by an adapter.

The application code looks up a ConnectionFactory instance from JNDI namespace and uses it to get EIS connections.

An implementation class for ConnectionFactory is required to implement `java.io.Serializable` and `javax.resource.Referenceableinterfaces` to support JNDI registration.

ADK Implementation

The ADK provides ConnectionFactoryImpl, a concrete implementation of the `javax.resource.cci.ConnectionFactory` interface that provides the following functionality:

- Access to the ADK logging framework.
- Access to adapter metadata.
- Implementation of the `getConnection()` method.

Typically, you will not need to extend this class and can use it outright.

ConnectionMetaData

`javax.resource.cci.ConnectionMetaData`

`ConnectionMetaData` provides information about an EIS instance connected through a Connection instance. A component calls the method `Connection.getMetaData` to get a `ConnectionMetaData` instance.

ADK Implementation

By default, the ADK provides an implementation of this class via the `com.bea.adapter.spi.AbstractConnectionMetaData` class. You will need to extend this abstract class and implement its four abstract methods for your adapter.

ConnectionSpec

`javax.resource.cci.ConnectionSpec`

ConnectionSpec is used by an application component to pass connection request-specific properties to the `ConnectionFactory.getConnection()` method.

It is recommended that you implement the ConnectionSpec interface as a JavaBean so that it can support tools. The properties on the ConnectionSpec implementation class must be defined through the getter and setter methods pattern.

The CCI specification defines a set of standard properties for an ConnectionSpec. The properties are defined either on a derived interface or an implementation class of an empty ConnectionSpec interface. In addition, an adapter may define additional properties specific to its underlying EIS.

ADK Implementation

Since the ConnectionSpec implementation must be a JavaBean, the ADK does not supply an implementation for this class.

InteractionSpec

`javax.resource.cci.InteractionSpec`

An InteractionSpec holds properties for driving an interaction with an EIS instance. It is used by an interaction to execute the specified function on an underlying EIS.

The CCI specification defines a set of standard properties for an InteractionSpec. An InteractionSpec implementation is not required to support a standard property if that property does not apply to its underlying EIS.

The InteractionSpec implementation class must provide getter and setter methods for each of its supported properties. The getter and setter methods convention should be based on the JavaBeans design pattern.

The InteractionSpec interface must be implemented as a JavaBean in order to support tools. An implementation class for InteractionSpec interface is required to implement the java.io.Serializable interface.

The Interaction spec contains information that is not in Record but helps determine what EIS function to invoke.

The standard properties are described in Table 6-5:

Table 6-5 Standard InteractionSpec Properties

Property	Description
`FunctionName`	Name of an EIS function
`InteractionVerb`	Mode of interaction with an EIS instance: `SYNC_SEND`, `SYNC_SEND_RECEIVE`, `SYNC_RECEIVE`
`ExecutionTimeout`	The number of milliseconds an Interaction will wait for an EIS to execute the specified function

The following standard properties are used to give hints to an interaction instance about the `ResultSet` requirements:

- `FetchSize`
- `FetchDirection`
- `MaxFieldSize`
- `ResultSetType`
- `ResultSetConcurrency`

A CCI implementation can provide additional properties beyond that described in the InteractionSpec interface.

Note: The format and type of the additional properties is specific to an EIS and is outside the scope of the CCI specification.

ADK Implementation

The ADK contains a concrete implementation of `javax.resource.cci.InteractionSpec` called InteractionSpecImpl. This interface provides a base implementation for you to extend by using getter and setter methods for the standard interaction properties described in Table 6-5.

LocalTransaction

`javax.resource.cci.LocalTransaction`

The LocalTransaction interface is used for application-level local transaction demarcation. It defines a transaction demarcation interface for resource manager local transactions. The system contract level LocalTransaction interface (as defined in the `javax.resource.spi package`) is used by the container for local transaction management.

A local transaction is managed internal to a resource manager. There is no external transaction manager involved in the coordination of such transactions.

A CCI implementation can (but is not required to) implement the LocalTransaction interface. If the LocalTransaction interface is supported by a CCI implementation, then the method Connection.getLocalTransaction should return a LocalTransaction instance. A component can then use the returned LocalTransaction to demarcate a resource manager local transaction (associated with the Connection instance) on the underlying EIS instance.

The `com.bea.adapter.spi.AbstractLocalTransaction` class also implements this interface.

For more information on local transactions, see “Transaction Demarcation.”

Record

`javax.resource.cci.Record`

The `javax.resource.cci.Record` interface is the base interface for representing an input or output to the `execute()` methods defined on an Interaction. For more information on the `execute()` methods, see "execute() Version 1" and "execute() Version 2."

A MappedRecord or IndexedRecord can contain another Record. This means that you can use `MappedRecord` and `IndexedRecord` to create a hierarchical structure of any arbitrary depth. A basic Java type is used as the leaf element of a hierarchical structure represented by a MappedRecord or IndexedRecord.

The Record interface can be extended to form one of the representations shown in Table 6-6:

Table 6-6 Record Interface Representations

Representation	Description
MappedRecord	A key-value pair based collection representing a record. This interface is based on the `java.util.Map`
IndexedRecord	An ordered and indexed collection representing a record. This interface is based on the `java.util.List`.
JavaBean based representation of an EIS abstraction	An example is a custom record generated to represent a purchase order in an ERP system.
`javax.resource.cci.ResultSet`	This interface extends both java.sql.ResultSet and javax.resource.cci.Record. A ResultSet represents tabular data.

Assuming the adapter implements a CCI interface, the next consideration is the record format for a service. A service has a request record format and a response record format. The request record provides input to the service and the response record provides the EIS response.

ADK Implementation

The ADK focuses on helping you implement an XML-based record format in the CCI layer. To this end, the ADK provides the DocumentRecord class. In addition, you can use BEA's schema toolkit to develop schemas to describe the request and response documents for a service.

The ADK provides `RecordImpl`, a concrete implementation of the `javax.resource.cci.Record` interface that provides getter and setter methods for record name and description.

If an adapter provider wants to use an XML-based record format (which is highly recommended), the ADK also provides the `com.bea.adapter.cci.AbstractDocumentRecordInteraction` class. This class ensures that the client passes DocumentRecord objects. In addition, this class provides convenience methods for accessing content in a `DocumentRecord`.

ResourceAdapterMetaData

`javax.resource.cci.ResourceAdapterMetaData`

The interface `javax.resource.cci.ResourceAdapterMetaData` provides information about capabilities of an adapter implementation. A CCI client uses a `ConnectionFactory.getMetaData` to get metadata information about the adapter. The `getMetaData()` method does not require establishment of an active connection to an EIS instance. The ResourceAdapterMetaData interface can be extended to provide more information specific to an adapter implementation.

Note: This interface does not provide information about an EIS instance that is connected through the adapter.

ADK Implementation

The ADK provides ResourceAdapterMetaDataImpl that encapsulates adapter metadata and provides getters and setters for all properties.

Step 5: Deploying the Adapter

After implementing the SPI and CCI interfaces for an adapter, you deploy it. To deploy your adapter, you need to complete these tasks:

- Update the `ra.xml` file
- Create the `.rar` File
- Deploy the `.rar` File

This section describes these tasks.

Step 5a: Update the ra.xml File

To deploy an adapter, you first must supply an adapter descriptor file. This file must be included in the adapter's `.rar` file as `META-INF/ra.xml`. The ADK provides a base implementation for this file with the sample adapter. If the adapter provider extends AbstractManagedConnectionFactory, you will need to supply the following properties in the `ra.xml` file:

Table 6-7 RA.XML Properties

Property	Example

LogLevel	This property specifies the log verbosity for this connection factory. Possible values are: `DEBUG`, `INFO`, `WARN`, `ERROR`, and `AUDIT`; for example:
	`<config-property>`
	`<config-property-name>LogLevel</config- property-name>`
	`<config-property-type>java.lang.String </config-property-type>`
	`<config-property-value>WARN</config- property-value>`
	`</config-property>`

Table 6-7 RA.XML Properties

Property	Example

LanguageCode	This property specifies the desired language code for log messages. This property is used with the CountryCode property to identify the correct message bundle from which to retrieve messages.
	
	`<config-property>`
	
	`<config-property-name>LanguageCode</config-property-name>`
	
	`<config-property-type>java.lang.String</config-property-type>`
	
	`<config-property-value>en</config-property-value>`
	
	`</config-property>`

CountryCode	This property specifies the desired country code for log messages.
	
	`<config-property>`
	
	`<config-property-name>CountryCode</config-property-name>`
	
	`<config-property-type>java.lang.String</config-property-type>`
	
	`<config-property-value>US</config-property- value>`
	
	`</config-property>`

MessageBundleBase	This property specifies the base name for all message bundles supplied with an adapter. The ADK always uses the adapter logical name for its sample adapters. However, you are free to chose your own naming convention for message bundles.
	
	`<config-property>`
	
	`<config-property-name>MessageBundleBase</config-property-name>`
	
	`<config-property-type>java.lang.String</config-property-type>`
	
	`<config-property-value>sample</config-property-value>`
	
	`</config-property>`

Table 6-7 RA.XML Properties

Property	Example

LogConfigFile	This property specifies the log configuration file for the adapter.
	`<config-property>`
	`<config-property-name>LogConfigFile</config-property-name>`
	`<config-property-type>java.lang.String</config-property-type>`
	`<config-property-value>sample.xml</config-property-value>`
	`</config-property>`

RootLogContext	This property specifies the root log context. Log context helps categorize log messages according to modules in a program. The ADK uses the adapter logical name for the root log context so that all messages from a specific adapter will be categorized accordingly.
	`<config-property>`
	`<config-property-name>RootLogContext </config-property-name>`
	`<config-property-type>java.lang.String</config-property-type>`
	`<config-property-value>sample</config-property-value>`
	`</config-property>`

AdditionalLogContext	This property allows the user to add additional log context information to help categorize messages. Typically, this is the name of the associated application view.
	`<config-property>`
	`<config-property-name>AdditionalLogContext</config-property- name>`
	`<config-property-type>java.lang.String</config-property-type>`
	`</config-property>`

You can see an example of a functioning `ra.xml` file for either a DBMS adapter or an e-mail adapter in one of the following locations:

- `<WLAI_HOME>\dev\email\src\rar\META-INF\`
- `<WLAI_HOME>\dev\dbms\src\rar\META-INF\`

Or, you can see an example for the cloned sample adapter at:

`<WLAI_HOME>\dev\sample\src\rar\META-INF\`

Step 5b: Create the weblogic-ra.xml File

The `weblogic-ra.xml` file is a deployment descriptor that contains essentially the same information as the ConnectionFactory descriptor, but in the form expected by the WebLogic J2EE engine. This file also contains a reference back to the "base" `.rar` deployment for the adapter. This reference allows the new `.rar` to use the code and other resources deployed in the base deployment, and frees it from containing itself. The reference to the base deployment also allows the new `.rar` to inherit any pool or connection parameters specified for base deployment.

When a user defines and starts to deploy an application view, a ConnectionFactory descriptor is created that contains the EIS connection parameters and connection pools parameters that the user entered. The ConnectionFactory descriptor also contains the JNDI name to which the new connection factory will be bound. This name is automatically generated based on the application view's qualified name. This ConnectionFactory descriptor is then deployed to the Application Integration engine, which, in turn, creates a new `.rar` file containing only a `weblogic-ra.xml` file.

A version of the `weblogic-ra.xml` file is included with the sample adapter at

`<WLAI_HOME>\dev\sample\src\rar\META-INF\`

When you run the GenerateAdapterTemplate utility, a version of this file tailored for you new adapter will be created.

You can see an example of a `weblogic-ra.xml` file for a DBMS adapter at:

`<WLAI_HOME>\dev\dbms\src\rar\META-INF\`

Step 5c: Create and Deploy the .rar File

Class files, logging configuration, and message bundle(s) should be bundled into a `.jar` file. This `.jar` file and `META-INF/ra.xml` should then be bundled into `.rar` file. The Ant `build.xml` file demonstrates how to properly construct the `.rar` file. See "Step 2c: Setting Up the Build Process." Listing 6-26 shows an example of a `.rar` file:

Listing 6-26

```
<Application Deployed="true"
Name="BEA_WLS_SAMPLE_ADK_WLSRAR.rarPath="d:\wlai\dev\sample\lib">

     <ResourceAdapterComponent

      Description="__BEA_Sample_Description__"

      Name="BEA_WLS_SAMPLE_ADK_WLSRAR" Targets="myserver"
URI="BEA_WLS_SAMPLE_ADK_WLSRAR.rar"/>

</Application>
```

Deploy the `.rar` file into the J2EE-compliant container in the application server. The deployment procedure will be different on every server.

Step 6: Testing the Adapter

The ADK provides a test harness that leverages JUnit, an open-source tool for unit testing. You can find more information on JUnit at http://`www.junit.org`.

`com.bea.adapter.test.TestHarness` does the following:

- Reads a properties file containing test configuration information.
- Initializes the log toolkit.
- Initializes JUnit TestSuite.

- Loads test classes and executes them using JUnit.
- Allows you to test code off-line and outside of WebLogic 6.0.

Using the Test Harness

To use the test harness in the ADK, complete the following steps:

1. Create a class that extends `junit.framework.TestCase`. The class must provide a static method named suite that returns a new `junit.framework.TestSuite`. See the sample adapter for details.
2. Implement test methods; name of methods should begin with "test"
3. Create/alter the `test.properties` in the project directory (if you clone the sample adapter, then your adapter will already have a base test.properties in the project directory). The properties file should contain any configuration properties needed for your test case.
4. Invoke the test using Ant. Your Ant `build.xml` file will need a test target that invokes the `com.bea.adapter.test.TestHarness` class with the properties file for your adapter. For example, the sample adapter uses the Ant target shown in Listing 6-27:

Listing 6-27 Ant Target Specified in the Sample Adapter

```
<target name='test' depends='packages'>
  <java classname='com.bea.adapter.test.TestHarness'>
     <arg value='-DCONFIG_FILE=test.properties'/<classpath
refid='CLASSPATH'/>
  </java>
</target>
```

This target invokes the JVM with main class `com.bea.adapter.test.TestHarness` using the classpath established for the sample adapter and passes the command-line argument:

```
-DCONFIG_FILE=test.properties
```

Test Case Extensions Provided by the ADK

The sample adapter ships with two basic TestCase extensions:

- `sample.spi.NonManagedScenarioTestCase`
- `sample.event.OfflineEventGeneratorTestCase`

sample.spi.NonManagedScenarioTestCase

NonManagedScenarioTestCase allows you to test your SPI and CCI classes in a non-managed scenario. Specifically, this class tests the following:

- Initialization of the ManagedConnectionFactory implementation.
- Serialization/De-serialization of the ManagedConnectionFactory instance.
- Opening a connection to the EIS.
- Closing a connection to the EIS; you can make sure all associated resources are getting closed when a connection is closed.

sample.event.OfflineEventGeneratorTestCase

`sample.event.OfflineEventGeneratorTestCase` allows you to test the inner workings of your event generator outside of the WebLogic Server. Specifically, this class tests the following for the event generator:

- It simulates the event router and instantiates a new instance of the adapter's event generator.
- It passes the `test.properties` to the event generator for initialization; this allows you to test your initialization logic.
- It refreshes the event generator randomly; this allows you to test your `setupNewTypes()` and `removeDeadTypes()` methods.
- It receives event postings and displays them to the log file for the adapter.

sample.client.ApplicationViewClient

`sample.client.ApplicationViewClient` offers an additional way of test your adapter. This class is a Java program that demonstrates how to invoke a service and listen for an event on an application view. The Ant `build.xml` provides the "client" target to allow you to use the ApplicationViewClient program. Executing `ant client` will provide the usage for the program. To see an example of `sample.client.ApplicationViewClient`, go to `<WLAI_HOME>/dev/sample/src/sample/client`.

Note: `sample.client.ApplicationViewClient` is not integrated with the test harness.

CHAPTER

7 Developing an Event Adapter

Event adapters propagate information from an EIS into the Application Integration environment. These types of adapters can be described as publishers of information. On the Application Integration integration platform, all event adapters perform the following three functions:

- They respond to “events” that occur inside the running EIS and extract data about the event from the EIS into the adapter.
- They transform event data from the EIS specific format to an XML document that conforms to the XML schema for the event. The XML schema is based on metadata in the EIS.
- They propagate the event into the Application Integration environment by using the event router.

The Application Integration implements the aspects of these three functions that are generic across all event adapters. You only need to focus on the EIS specific aspects of your adapter.

This section contains information on the following subjects:

- Event Adapters in the Runtime Environment
- The Flow of Events
- Step 1: Development Considerations
- Step 2: Configuring the Development Environment
  - Step 2a: Set up the File Structure
  - Step 2b: Assign the Adapter Logical Name

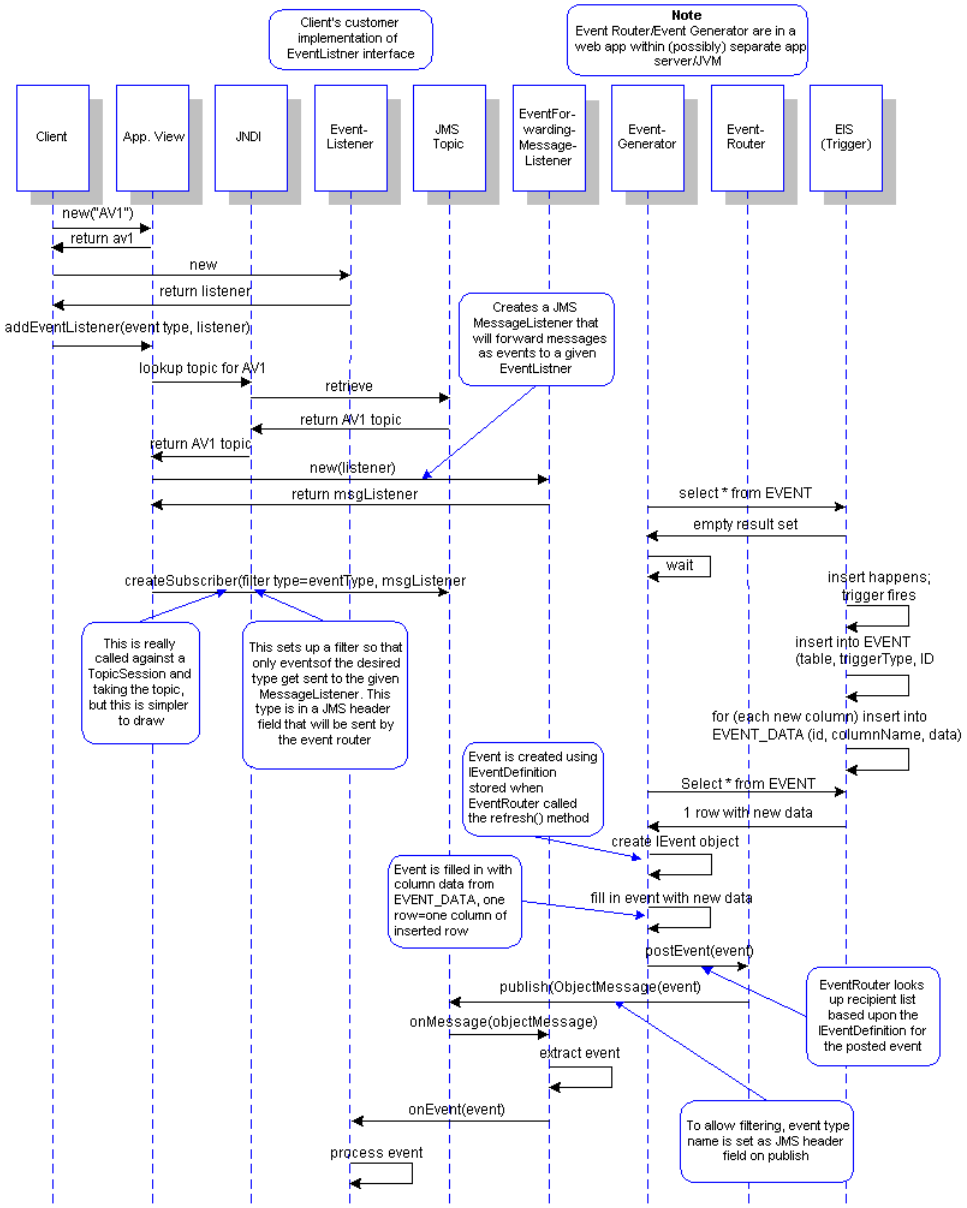
- Step 2c: Set Up the Build Process
- Step 2d: Create the Message Bundle
- Step 2e: Configure Logging

- Step 3: Implementing the Adapter
  - Step 3a: Create an EventGenerator
  - Step 3b: Implement the Data Transformation Method
- Step 4. Deploying the Adapter
- Step 5: Testing the Adapter

Event Adapters in the Runtime Environment

The behavior of an Event in the runtime environment is depicted in Figure 7-1.

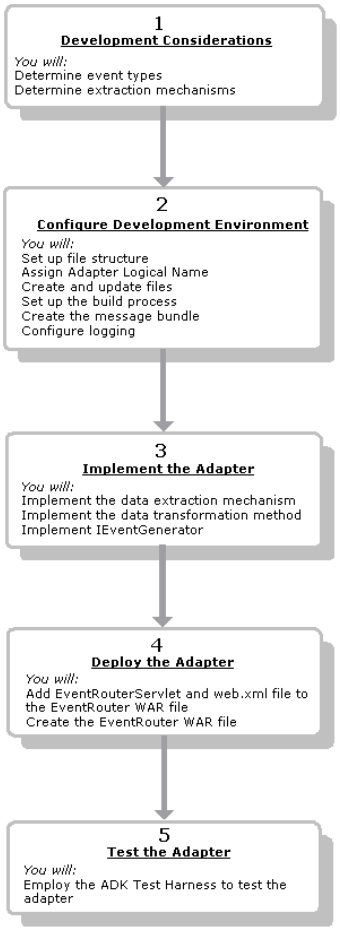
Figure 7-1 Event Adapters in the Runtime Environment



The Flow of Events

Figure 7-2 outlines the steps required to develop an Event Adapter.

Figure 7-2 Event Adapter Flow of Events



Step 1: Development Considerations

These are the items you need to consider before commencing with event adapter development. The Adapter Setup Worksheet will provide much of this information. See "Adapter Setup Worksheet."

1. Determine the event types.

   You need to identify what exactly comprises the event:

   - What will its contents be?
   - How will it be defined in the XML schema?
   - What will trigger it?

2. Determine the data extraction method.

   Next, you need to decide which method of data extraction will be used:

   - "Push," wherein the EIS notifies the adapter of an event.
   - "Pull," where the adapter polls the EIS and pulls event data from it.

   Use the pull method when your adapter needs to poll the EIS to determine a change-of-state. Use a push event when you want to implement an event generation that works more like a publish/subscribe model.

Step 2: Configuring the Development Environment

This step describes the processes you must complete to prepare your computer for adapter development.

Step 2a: Set up the File Structure

The file structure necessary to build an event adapter is the same as that required for service adapters. See "Step 2a: Set up the File Structure" in "Developing a Service Adapter."

Step 2b: Assign the Adapter Logical Name

Next, you need to assign the adapter's logical name. By convention, this name is comprised of the vendor name, the type of EIS connected to the adapter, and the version number of the EIS and is expressed as *vendor_EIS-type_EIS version*. For example, `BEA_WLS_SAMPLE_ADK`, where:

- `BEA` is the vendor
- `WLS` is the EIS-type
- `SAMPLE_ADK` is the EIS version

Step 2c: Set Up the Build Process

The Application Integration employs a build process based upon Ant, a 100% pure Java-based build tool. For more information on Ant, please see "Ant-Based Build Process." For more information on using Ant, see http://jakarta.apache.org/ant/index.html.

The sample adapter shipped with BEA WebLogic Application Integration (located in `<WLAI_HOME>\dev\bin\ sample`) contains the file `build.xml`. This is the Ant build file for the sample adapter. It contains the tasks needed to build a J2EE-compliant adapter. Running the GenerateAdapterTemplate utility to clone a development tree for your adapter creates a `build.xml` file specifically for that adapter. This will free you from having to customize the sample `build.xml` and will ensure that the code is correct. For information on using the GenerateAdapterTemplate utility, see "Creating a Custom Development Environment."

For more information on the build process, see "Step 2c: Setting Up the Build Process" in "Developing a Service Adapter."

Step 2d: Create the Message Bundle

Any message destined for the end-user should be placed in a message bundle. The message bundle is simply a `.properties` text file that contains key=value pairs that allow you to internationalize messages. When a locale and national language are specified at runtime, the contents of the message is interpreted, based upon the key=value pair, and the message is presented to the user in the correct language for his or her locale.

For instructions on creating a message bundle, please refer to the JavaSoft tutorial on internationalization at:

`http://java.sun.com/docs/books/tutorial/i18n/index.html`

Step 2e: Configure Logging

The final step in configuring your development environment is to configure logging. Before you begin, read more about logging in "Concepts." Logging is accomplished using the logging tool log4j, developed as part of the Apache Jakarta project. For information on using this tool, please see "The ADK Logging Toolkit."

Create an Event Generation Logging Category

For event adapters, you will need to create a logging category specifically for event generation (for more information on logging categories, see "Message Categories"). Edit the logging configuration file for the specific adapter (*`adapter_logical_name`*`.xml` in the `/src/` directory for that adapter) by adding the following line:

Listing 7-1 Sample Code Creating an Event Generation Logging Category

```
<category name='BEA_WLS_SAMPLE_ADK.EventGenerator' class='com.bea.
logging.LogCategory'>

</category>
```

By not setting any specific parameters for this category, it will inherit all of the parent category's property settings. In this example, the parent category is `BEA_WLS_SAMPLE_ADK`.

Step 3: Implementing the Adapter

Implementing an event adapter is a two-step process. You need to:

- Create an EventGenerator. This process implements the data extraction method (that is, will you extract data by a push or a pull mechanism) and the IEventGenerator interface. This interface is used by the EventRouter to drive the event generation process.
- Implement the data transformation method.

This section shows you how to accomplish these tasks.

Step 3a: Create an EventGenerator

The event generation provides adapter with a mechanism to either receive notification from an EIS or poll an EIS for the specific occurrence of an event. The event generation provided by the BEA WebLogic Application Integration engine is very powerful in that a single EventGenerator can support multiple types of events. An event type is defined by the configuration properties for an event.

Typically event properties are defined by the properties associated with an event at designtime. When configuring an event adapter, the adapter may have one or more web pages that it uses to collect event properties. These properties are saved with the application view Descriptor and passed back to the event at runtime. The Application Integration engine uses the properties and the source application view to determine how to route back to the listeners. For instance, two separate deployments of the same EventGenerator with identical properties will result in only a single IEventDefinition being created by the Application Integration engine. Whereas, a single IEventDefinition will be created for every deployment of a single event adapter where

the properties are different. It is the responsibility of the EventGenerator to determine which IEventDefinition to use in the routing process. This is typically done based on property values and specific event occurrences.

The IEventDefinition objects are used by your implementation of the EventGenerator to route specific events back to their listener. As discussed before, the Application Integration engine will create IEventDefinition objects for deployed application views containing events. You will use the IEventDefinition objects to extract specific properties regarding the deployment of an application view, or to access schema and routing objects. You need to employ these attributes when routing an event.

How the Data Extraction Mechanism is Implemented

Application Integration supports two mechanisms for data extraction:

- Pull event generation. A state change is recognized when the object generating events pushes a notification to the EventGenerator. When the Push Event generator receives the event, the Application Integration engine then routes to a deployed application view. The push EventGenerator uses a publish/subscribe model.
- Push event generation, which is used when polling is necessary to accomplish the determination of a state having changed. A process continually queries an object until it has determined a change in state, at which point it creates an event, which the BEA WebLogic Application Integration engine the routes to a deployed application view.

The “Pull” Mechanism

The mechanism relies on a polling technique to determine if an event has taken place. To implement a Pull scenario you must derive your EventGenerator from the AbstractPullEventGenerator in the `com.bea.adapter.event` package.

Note: `adk-eventgenerator.jar` file must be included in your `.war` make file. `adk-eventgenerator.jar` contains the ADK base classes required to implement an EventGenerator.

The ADK supplies several abstract methods in the AbstractPullEventGenerator that you must override in your implementation. These methods are described in Table 7-1.

Table 7-1 AbstractPullEventGenerator Methods

Method	Description
`postEvents()`	The postEvents method is called from the run method in the AbstractPullEventGenerator at an interval that is determined by the Event Router configuration files. The `postEvents()` method is where you add any polling and routing code. The postEvents method is the control method for the rest of your event generation, message transformation, and routing code.
setupNewTypes()	The `setupNewTypes()` method is used to preprocess any IEventDefinition object being deployed. Only valid new IEventDefinition objects are passed to the setupNewTypes method.
removeDeadTypes()	The removeDeadTypes method is used to handle any clean up required for IEventDefinition objects that are being un-deployed. The Application Integration engine calls removeDeadTypes when application views with associated events are being un-deployed.
`doInit()`	`doInit()` is called while the EventGenerator is being constructed. During the initialization process the EventGenerator can use pre-defined configuration values to setup the necessary state or connections for the event generation process.
`doCleanUpOnQuit()`	`doCleanUpOnQuit()` is called before ending the thread driving the event generation process. Use this method to free any resources allocated by your event generation process.

The “Push” Mechanism

The Push scenario uses notification to trigger the routing of an event. To implement the Push scenario you must derive your EventGenerator from the AbstractPushEventGenerator class in the `com.bea.adapter.event` package. There are several other supporting classes included in the event package. These classes are described in Table 7-2.

Note: `adk-eventgenerator.jar` must be included in your `.war` make file. `adk-eventgenerator.jar` contains the Application Integration base classes required to implement an EventGenerator.

Table 7-2 AbstractPushEventGenerator Classes

Class	Description
`AbstractPushEventGenerator`	The AbstractPushEventGenerator class contains the same abstract and concrete methods as the AbstractPullEventGenerator. These methods are intended to be used in the same manner as the AbstractPullEventGenerator implementation. See Table 7-1 for a list of these methods and responsibilities.
`IPushHandler`	The IPushHandler is an interface provided primarily to abstract the generation of an event from the routing of an event; however, it is not required to implement a Push scenario. The IPushHandler is intended to be tightly coupled with the PushEventGenerator. It is the PushEventGenerator that will initialize, subscribe, and clean up the PushHandler implementation. The IPushHandler provides a simple interface to abstract the generation logic. The interface provides methods to initialize, subscribe to Push events, and clean up resources.
`PushEvent`	The PushEvent is an event object derived from `java.util.EventObject`. The PushEvent is intended to wrap an EIS notification and be sent to any IPushEventListener objects.
`EventMetaData`	The EventMetaData class is intended to wrap any data necessary for event generation. The EventMetaData class is passed to the IPushHandler on initialization. To see a sample usage for these objects refer to the email sample code.

How the EventGenerator is Implemented

An EventGenerator implementation typically follows this flow of control:

- `doInit();` This method creates and validates connections to the EIS.
- `setupNewTypes();` This method processes IEventDefinition objects creating any required structures for processing.
- `postEvents();` This method iteratively invokes one of the two data extraction mechanisms:
  - Push: Poll the EIS for an Event. If the Event exists, determine which IEventDefinition objects will receive the event. Transform event data into an IDocument object using the associated schema. Route the IDocument object using the IEvent associated with the IEventDefinition object.
  - Pull: When notified of an event the postEvents method will extract the event data from the PushEvent object and transform the event data to an IDocument object. The IDocument object is created based on the schema associated with the event adapter. When the IDocument contains the necessary event data it is routed to the correct IEventDefinition objects.
- `removeDeadTypes();` This method removes the dead IEventDefinition objects from any data structures being used for event processing. Free any resources associated. IEventDefinition objects are considered "dead" when the application view is undeployed.
- `doCleanUpOnQuit();` This method removes any resources allocated during event processing.

The following is a series of code samples that implement an EventGenerator with a Pull mechanism.

Listing 7-2 shows the class declaration for the sample adapter's (Pull) EventGenerator.

Note: The AbstractPullEventGenerator implements a "run-able" interface; that is, it's a thread.

Listing 7-2 Sample Code Implementing a Pull Data Extraction Mechanism

```
public class EventGenerator
  extends AbstractPullEventGenerator
```

Sample EventGenerator

Listing 7-3 shows the simple constructor for an EventGenerator. You must invoke the parent's constructor so that the parent's members get initialized correctly. The listing then shows how the `doInit()` method receives configuration information from the `map` variable and validates the parameters. The sample contains any parameters associated with the EventGenerator at designtime.

Listing 7-3 Sample Constructor for an EventGenerator

```
public EventGenerator()
{
super();
}
  protected void doInit(Map map)
    throws java.lang.Exception
  {
    ILogger logger = getLogger();
    m_strUserName = (String)map.get("UserName");
        if (m_strUserName == null || m_strUserName.length() == 0)
    {
      String strErrorMsg =
logger.getI18NMessage("event_generator_no_UserName");
      logger.error(strErrorMsg);
      throw new IllegalStateException(strErrorMsg);
    }
    m_strPassword = (String)map.get("Password");
    if (m_strPassword == null || m_strPassword.length() == 0)
    {
```

```
      String strErrorMsg = logger.getI18NMessage
("event_generator_no_Password");
      logger.error(strErrorMsg);
      throw new IllegalStateException(strErrorMsg);
    }
```

`postEvents()` is called from the run method of our parent class, as shown in Listing 7-4. This method polls the EIS to determine when a new event occurs. This method will be invoked at a fixed interval, which is defined in the `web.xml` file for the EventRouter.

Listing 7-4 Sample Code Implementation of the postEvents() Method

```
 */  protected void postEvents(IEventRouter router)
    throws java.lang.Exception
  {
    ILogger logger = getLogger();
    // TODO: a real adapter would need to call into the EIS to
    // determine ifany new events occured since the last time this
    // method was invoked. For the sake of example, we'll just post
    // a single event every time this method gets invoked...The
    // event data will be the current time on the
    // system formatted according to the event definition...we'll
   // look for several event types...
    Iterator eventTypesIterator = getEventTypes();
    if (eventTypesIterator.hasNext())
    {
      do
      {
       // The event router is still interested in this type of event
```

```
        IEventDefinition eventDef = (IEventDefinition)
eventTypesIterator.next();

       logger.debug("Generating event for " + eventDef.getName());

        // Create a default event (just blank/default data)

        IEvent event = eventDef.createDefaultEvent();

        // Get the format for the event

       java.util.Map eventPropertyMap = eventDef.getPropertySet();

        String strFormat = (String)eventPropertyMap.get("Format");

        if( logger.isDebugEnabled() )

          logger.debug("Format for event type '"+eventDef.
getName()+"' is '"+strFormat+"'");

          java.text.SimpleDateFormat sdf =

          new java.text.SimpleDateFormat(strFormat);

          IDocument payload = event.getPayload();

        payload.setStringInFirst("/SystemTime", sdf.format(new
Date()));

        // let's log an audit message for this...

        try

        {

         logger.audit(toString() + ": postEvents >>> posting event
["+payload.toXML()+"] to router");

        }

        catch (Exception exc)

        {

          logger.warn(exc);

        }

        // This call actually posts the event to the IEventRouter

        router.postEvent(event);

      } while (eventTypesIterator.hasNext());
```

```
    }
  }// end of postEvents
```

A real adapter would need to call into the EIS to determine if any new events occurred since the last time this method was invoked. You can see a concrete example of this in the DBMS adapter included with the ADK. Refer to the `postEvent()` method in `DbmsEventGeneratorWorker.java`, which is in:

`<WLAI_HOME>dev/dbms/com/bea/adapter/dbms/event/`

Adding New Event Types

`setupNewTypes()` gets called during refresh to handle any new event types. This allows us to perform any setup we need to handle a new type. The parent class has already sanity-checked the `listOfNewTypes()` and logged it; so you don't need to do that here.

Listing 7-5 Sample Code Showing the Template for setupNewTypes()

```
protected void setupNewTypes(java.util.List listOfNewTypes)
{
  Iterator iter = listOfNewTypes.iterator();
  while (iter.hasNext())
  {
    IEventDefinition eventType = (IEventDefinition)iter.next();
   }
}
```

Removing Event Types for Application Views that are Undeployed

`removeDeadTypes()` is called during refresh to handle any event types for application views that have been undeployed. You will need to perform a cleanup process to ensure that this event type is no longer handled, such as closing resources needed to handle this specific event type. Listing 7-6 shows how `removeDeadTypes()` is implemented.

Listing 7-6 Sample Code Showing the Template for removeDeadTypes()

```
protected void removeDeadTypes(java.util.List listOfDeadTypes)
{
     Iterator iter = listOfDeadTypes.iterator();
  while (iter.hasNext())
  {
    IEventDefinition eventType = (IEventDefinition)iter.next();
```

Removing Resources

Finally, `doCleanUpOnQuit()` gets called when the EventGenerator is shutting down, This method removes any resources allocated during event processing. The sample adapter stubs in this method. The template for implementing this method is shown in Listing 7-7.

Listing 7-7 Sample Code Showing doCleanUpOnQuit() Method Call

```
protected void doCleanUpOnQuit()
  throws java.lang.Exception
{
  ILogger logger = getLogger();
  logger.debug(this.toString() + ": doCleanUpOnQuit");
}
```

```
}
```

Step 3b: Implement the Data Transformation Method

Data transformation is the process of taking data from the EIS and transforming it into an XML schema that can be read by the application server. For each event, a schema will define what the XML output looks like. This is accomplished by using the SOM and Document class libraries. The following code listings show the data transformation sequence:

- Listing 7-8 shows the code used to transform data from the EIS into XML schema.
- Listing 7-9 shows the XML schema created by the code in Listing 7-8.
- Listing 7-10 shows the valid XML document created by the schema shown in Listing 7-9.

Listing 7-8 Sample Code for Transforming EIS Data into XML Schema

```
SOMSchema schema = new SOMSchema();
SOMElement root = new SOMElement("SENDINPUT");
SOMComplexType mailType = new SOMComplexType();
root.setType(mailType);
SOMSequence sequence = mailType.addSequence();
SOMElement to = new SOMElement("TO");
to.setMinOccurs("1");
to.setMaxOccurs("unbounded");
sequence.add(to);
SOMElement from = new SOMElement("FROM");
from.setMinOccurs("1");
from.setMaxOccurs("1");
```

```
sequence.add(from);
SOMElement cc = new SOMElement("CC");
cc.setMinOccurs("1");
cc.setMaxOccurs("unbounded");
sequence.add(cc);
SOMElement bcc = new SOMElement("BCC");
bcc.setMinOccurs("1");
bcc.setMaxOccurs("unbounded");
sequence.add(bcc);
SOMElement subject = new SOMElement("SUBJECT");
subject.setMinOccurs("1");
subject.setMaxOccurs("1");
sequence.add(bcc);
SOMElement body = new SOMElement("BODY");
if (template == null)
   { body.setMinOccurs("1");
     body.setMaxOccurs("1");
   }else
     { Iterator iter = template.getTags();
       if (iter.hasNext())
       { SOMComplexType bodyComplex = new SOMComplexType();
         body.setType(bodyComplex);
         SOMAll all = new SOMAll();
         while (iter.hasNext())
         { SOMElement eNew = new SOMElement((String)iter.next());
           all.add(eNew);
         }//endwhile
```

```
          bodyComplex.setGroup(all);
       }//endif
     }//endif
sequence.add(body);
schema.addElement(root);
```

Listing 7-9 XML Schema Created by Code in Listing 7-8

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SENDINPUT">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="TO" maxOccurs="unbounded" type="xsd:
string"/>
        <xsd:element name="FROM" type="xsd:string"/>
        <xsd:element name="CC" maxOccurs="unbounded" type=
"xsd:string"/>
        <xsd:element name="BCC" maxOccurs="unbounded" type=
"xsd:string"/>
        <xsd:element name="BCC" maxOccurs="unbounded" type=
"xsd:string"/>
        <xsd:element name="BODY" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

Listing 7-10 Valid XML Document Created by Schema in Listing 7-9

```
</xsd:schema>
```

```
 <?xml version="1.0"?>

<!DOCTYPE SENDINPUT>

<SENDINPUT>

  <TO/>

  <FROM/>

  <CC/>

  <BCC/>

  <BCC/>

  <BODY/>

</SENDINPUT> <xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Step 4. Deploying the Adapter

To deploy the event adapter, you need to deploy the EventGenerator on a J2EE servlet engine outside of the BEA WebLogic Application Integration engine. To do this, you need to bundle the EventGenerator as a web application, `WEB-INF/web.xml`. Rather than create this file from scratch, we recommend that you use the GenerateAdapterTemplate utility to clone and customize a development environment for your adapter. Running this utility will modify the `web.xml` file in the sample adapter for use with the adapter you are developing. For more information on using GenerateAdapterTemplate, see “Creating a Custom Development Environment.”

Add EventRouterServlet to .war File

The EventRouter is a `.war` file used to deploy your EventGenerator and is required for any adapter supporting events. It will be the URL that you will use in the design-time interface to deploy the event adapter. Because it is a `.war` file, it requires an associated `web.xml` file for deployment.

The EventRouterServlet is responsible for initializing and starting the Application Integration event process. The `web.xml` file contains initialization parameters for the EventRouterServlet, some of which are shown in Table 7-11.

Listing 7-11 EventRouterServlet Element of web.xml

```
<servlet>

   <servlet-name>EventRouterServlet</servlet-name>

   <description>Provides access to event router within
      WLAI</description>

   <servlet-class>com.bea.wlai.event.EventRouterServlet</servlet-c
      lass>

   <!-- Or jsp-file -->

   <init-param>

      <param-name>eventGeneratorClassName</param-name>

      <param-value>email.event.PullEventGenerator</param-value>

      <description>The fully qualified class name of the
         IEventGenerator implementation class</description>

   </init-param>

   <init-param>

      <param-name>wlsUserID</param-name>

      <param-value>system</param-value>

      <description>The user id on the WLAI server that this event
         router will talk to to get schemas, etc.</description>

   </init-param>

   <init-param>

      <param-name>wlsPassword</param-name>

      <param-value>weblogic</param-value>

      <description>The password for wlsUserID</description>

   </init-param>
```

```
   .
   .
   .
   <load-on-startup>1</load-on-startup>
</servlet>
```

You need to assign values to several initialization parameters prior to creating the EventRouter. These parameters are described in Table 7-3.

Table 7-3 Event Router Initialization Parameters

Parameter	Description	Value
`EventRouterServlet`	Class name for the event router servlet. Do not change this value.	`com.bea.wlai.event.EventRouter-Servlet`
`eventGeneratorClassName`	Class name for the EventGenerator. This is the fully qualified class name for your EventGenerator.	`email.event.PushEventGenerator`
`wlsUserID`	User ID for Weblogic Server. This user ID will be used to connect to weblogic in order to retrieve the Schema's associated with this event adapter.	`system`
`wlsPassword`	Password for Weblogic Server.	`weblogic`
`userName`	This is the user name for the EIS the EventGenerator will be connecting to.	`username`

Table 7-3 Event Router Initialization Parameters (Continued)

Parameter	Description	Value
`password`	Password for the EIS connection.	`password`
`connectionURL`	URL describing the location for the EIS	`server.domain.com`
`RootLogContext`	Base log context. If no additional log context is specified this will be the context for the EventGenerator.	`adapterLogicName`
`AdditionalLogContext`	Used to further sub-categorize the logging messages. The additional log context can be used to identify logging messages from a specific context. The AdditionalLogContext is used in conjunction with the RootLogContext to identify a new Log Context; that is: `(root.additionalLog- Context)`	`sub-category`
`LogConfigFile`	The log configuration file for this adapter. The log configuration file contains an XML document with setup parameters for the log utility.	`filename.xml`
`MessageBundleBase`	Base name for message bundles. Base name and language code are used to identify a specific instance of a message bundle.	filename

Table 7-3 Event Router Initialization Parameters (Continued)

Parameter	Description	Value
`LanguageCode`	Preferred language code for log messages	en
`CountryCode`	Preferred country code for log messages.	US
`sleepCount`	Count between iterations. This is the time between calls to the postEvents method for the PullEventGenerator. This count is given in milliseconds.	5000

The `web.xml` file also contains servlet directives and security constraints. Refer to the WebLogic documentation for a description of these attributes.

Create the EventRouter .war file

To make the EventRouter `.war` file you must include the following `.jar` files in your `.war` file:

- `log4j.jar`
- `wlai-common.jar`
- `wlai-ejb-client.jar`
- `wlai-eventrouter.jar`
- `wlai-servlet-client.jar`
- `logtoolkit.jar`
- `adk-eventgenerator.jar`

Also add any other `.jar` file upon which your EventGenerator is dependent. These are specified in the `<eventrouter>` target in the `build.xml` file, as shown in Listing 7-12.

Listing 7-12 Adding .jar Files to the `eventrouter` Target of build.xml

```
<target name='eventrouter_war' depends='jar,eventrouter_jar'>

   <delete file='${LIB_DIR}/${EVENTROUTER_WAR_FILE}'/>

   <delete dir='${SRC_DIR}/eventrouter/WEB-INF/lib'/>

   <war warfile='${LIB_DIR}/${EVENTROUTER_WAR_FILE}
'webxml='${SRC_DIR}/eventrouter/WEB-INF/web.xml'>

     <fileset dir='${PROJECT_DIR}' includes='version_info.xml'/>

     <fileset dir='${SRC_DIR}/eventrouter' excludes='WEB-INF/
web.xml'/>

       <lib dir='${LIB_DIR}' includes='${JAR_FILE},
${EVENTROUTER_JAR_FILE}'/>

       <lib dir='${WLAI_LIB_DIR}' includes='adk.jar,adk-
eventgenerator.jar,bea.jar,logtoolkit.jar,wlai-common.jar,wlai-ejb-client.jar,wlai-eventrouter.jar,wlai-servlet-client.jar,xmltoolkit.jar'/>

     <lib dir='${RESOURCE_DIR}/log4j' includes='log4j.jar'/>

   </war>

</target>
```

Step 5: Testing the Adapter

You can test the adapter by using the adapter test harness provided with the Application Integration. See "Step 6: Testing the Adapter" in "Developing a Service Adapter" for a complete description of this tool and instructions for using it.

CHAPTER

8 Developing a Design-Time GUI

The ADK's design-time framework provides the tools you will use to build the web-based GUI that adapter users need to define, deploy, and test their application views. Although each adapter has EIS-specific functionality, all adapters require a GUI for deploying application views. The design-time framework minimizes the effort required to create and deploy these interfaces, primarily by using these two components:

- A web application component that allows you to build an HTML-based GUI by using Java Server Pages (JSP). This component is augmented by tools such as the JSP templates and tag library and the JavaScript library.
- The abstractDesignTimeRequestHandler class, which provides a simple API for deploying, undeploying, copying, and editing application views on an Application Integration server.

This section includes information on the following subjects:

- Introduction to Design-Time Form Processing
- Design-Time Features
  - Java Server Pages
  - JSP Templates
  - The ADK Tag Library
  - JavaScript Library
  - The Application View
- File Structure

Introduction to Design-Time Form Processing

There are a variety of approaches to processing forms using Java Servlets and JSPs. The basic requirements of any form processing approach are:

1. Display an HTML form.

   To accomplish this task, you must:

   - Generate the form layout using HTML.
   - Indicate to the user which fields are mandatory.
   - Prepopulate fields with defaults, if any.

2. When the user submits the form data, validate the field values in the HTTP request.

   To accomplish this task, you must:

   - Supply logic to determine if all mandatory fields have a value.
   - For each value submitted, validate it against a set of constraints; for example, seeing if an age field is a valid integer between 1 and 120.

3. If any field values are invalid, the form must be redisplayed to the user with an error message next to each erroneous field on the form. The error message should be localized for the user's preferred locale if the web application supports multiple locales. In addition, the user's last input should be redisplayed so they do not have to re-input any valid information. The web application should continue with Step 2 and loop as many times as needed until all fields submitted are valid.

4. Once all fields have passed coarse-grained validation, the form data must be processed. While processing the form data, an error condition may be encountered that does not relate to individual field validation, such as a Java exception. The form will need to be re-displayed to the user with a localized error message at the top of the page. As with step 3, all input fields should be saved so the user does not have to re-enter any valid information.

   To accomplish this task, the web application developer must:

   - Determine which object or method implements the form processing API.

- Determine how and when to advance the user to the next page in the web application.

5. If the form processing succeeds, the next page in the web application is displayed to the user.

Form Processing Classes

As you can imagine, or have experienced, implementing all these steps for every form in a web application is quite a tedious and error prone development process. The ADK Design Time Framework simplifies this process by using a Model-View-Controller paradigm. There are five classes involved in the form processing mechanism:

RequestHandler

`com.bea.web.RequestHandler`

This class provides HTTP request processing logic. This class is the Model component of the MVC-based mechanism. This object is instantiated by the ControllerServlet and saved in the HTTP session under the key `handler`. The ADK provides the `com.bea.adapter.web.AbstractDesignTimeRequestHandler`. This abstract base class implements functionality needed to deploy an application view that is common across all adapters. You will need to extend this class to supply adapter/EIS specific logic.

ControllerServlet

`com.bea.web.ControllerServlet`

This class is responsible for receiving an HTTP request, validating each value in the request, delegating the request to a RequestHandler for processing, and determining which page to display to the user. The ControllerServlet uses Java reflection to determine which method to invoke on the RequestHandler. The ControllerServlet looks for an HTTP request parameter named `doAction` to indicate the name of the method that implements the form processing logic. If this parameter is not available, the ControllerServlet does not invoke any methods on the RequestHandler.

The ControllerServlet is configured in the `web.xml` file for the web application. The ControllerServlet is responsible for delegating HTTP requests to a method on a RequestHandler. You don't need to provide any code to use the ControllerServlet. However, you must supply the initial parameters listed in Table 8-5:

ActionResult

com.bea.web.ActionResult

ActionResult encapsulates information about the outcome of processing a request. Also provides information to the ControllerServlet to help it determine the next page to display to the user.

Word and Its Descendants

`com.bea.web.validation.Word`

All fields in a web application require some validation. The `com.bea.web.validation.Word` and its descendants supply logic to validate form fields. If any fields are invalid, the Word object uses a message bundle to retrieve an internationalized/localized error message for the field. The ADK supplies the custom validators described in Table 8-1.

Table 8-1 Custom Validators for Word Object

Validator	Description
`Integer`	Determines if the value for a field is an integer within a specified range.
`Float/Double`	Determines if the value for a field is a floating point value within a specified range.
`Identifier`	Determines if the value for a field is a valid Java identifier.
`Perl 5 Regular Expression`	Determines if the value for a field matches a Perl 5 regular expression.
`URL`	Determines if the supplied value is a valid URL
`Email`	Determines if the supplied value contains a list of valid email addresses.

Table 8-1 Custom Validators for Word Object
(Continued)

Validator	Description
`Date`	Determines if the supplied value is a valid date using a specified date/time format

AbstractInputTagSupport and Its Descendants

`com.bea.web.tag.AbstractInputTagSupport`

The tag classes provided by the Web toolkit are responsible for:

- Generating the HTML for a form field and pre-populating its value with a default, if applicable.
- Displaying a localized error message next to the form field if the supplied value is invalid.
- Initializing a `com.bea.web.validation.Word` object and saving it in web application scope so that the validation object is accessible by the ControllerServlet using the form field's name.

Submit Tag

Additionally, the ADK provides a submit tag, such as:

```
<adk:submit name='xyz_submit' doAction='xyz'/>
```

This tag ensures the `doAction` parameter is passed to the ControllerServlet in the request. This results in the ControllerServlet invoking the xyz method on the registered RequestHandler.

Form Processing Sequence

This section discusses the sequence in which forms are processed. Figure 8-1 shows how forms are processed.

Prerequisites

Before forms can be processed, the following must occur:

1. When a JSP containing a custom ADK input tag is being written to the HTTP response object, the tag ensures that it initializes an instance of `com.bea.web.validation.Word` and places it into the web application scope, keyed by the input field name. This makes the validation object available to the ControllerServlet so that it can perform coarse-grained validation on an HTTP request prior to submitting the request to the RequestHandler. For example,

   ```
   <adk:int name='age' minInclusive='1' maxInclusive='120'
   required='true'/>
   ```

2. The HTML for this tag will be generated when the JSP engine invokes the `doStartTag()` method on an instance of `com.bea.web.tag.IntegerTagSupport`. The IntegerTagSupport instance will instantiate a new instance of `com.bea.web.validation.IntegerWord` and add it to web application scope under the key `age`. Consequently, the ControllerServlet can retrieve the IntegerWord instance from its ServletContext whenever it needs to validate a value for age. The validation will ensure that any value passed for age is greater than or equal to 1 and less than or equal to 120.

3. Lastly, the HTML form must also submit a hidden field named `doAction`. The value of this parameter is used by the ControllerServlet to determine the method on the RequestHandler that can process the form.

Following these prerequisites, the JSP form appears as shown in Listing 8-1:

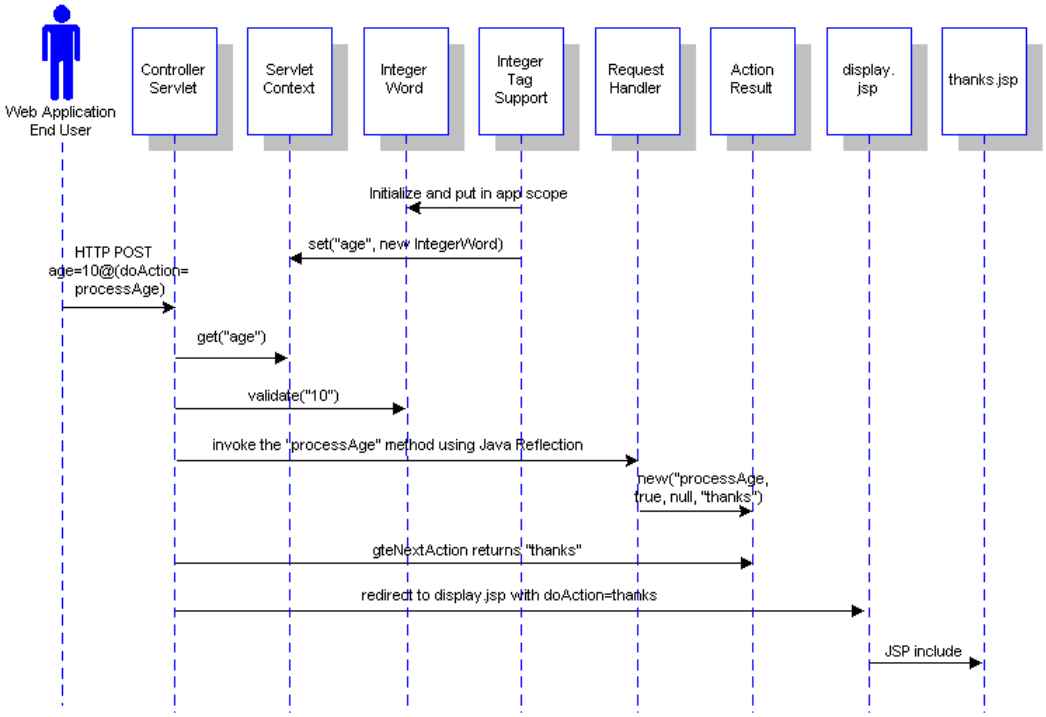
Listing 8-1 Sample JSP Form

```
<form method='POST' action='controller'>

   Age: <adk:int name='age' minInclusive='1' maxInclusive='120'
      required='true'/>

   <adk:submit name='processAge_submit' doAction='processAge'/>

</form>
```

Steps in the Sequence

The sequence diagram shown in Figure 8-1 illustrates the transactions that occur during form processing.

Figure 8-1 UI Form Processing Sequence Diagram



The sequence is as follows:

1. User submits the form with `age=10`, `doAction=processAge`.
2. ControllerServlet retrieves the age field from the HTTP request.
3. ControllerServlet retrieves a `com.bea.web.validation.Word` object from its ServletContext using key `age`. The object is an instance of `com.bea.web.validation.IntegerWord`.

4. ControllerServlet invokes the `validate()` method on the Word instance and passes `10` as a parameter.
5. The Word instance determines that the value 10 is greater than or equal to 1 and is less than or equal to 120. The Word instance returns true to indicate that the value is valid.
6. The ControllerServlet retrieves the RequestHandler from the session or creates it and adds it to the session as `handler`.
7. The ControllerServlet uses the Java Reflection API to locate and invoke the `processAge()` method on the RequestHandler. An exception is generated if the method does not exist. The method signature is:

   ```
   public ActionResult processAge(HttpServletRequest request)
   throws Exception
   ```

8. The RequestHandler processes the form input and returns an ActionResult object to indicate the outcome of the processing. The ActionResult contains information used by the ControllerServlet to determine the next page to display to the user. The next page information should be the name of another JSP or HTML page in your web application; for example, `thanks` would display the `thanks.jsp` page to the user.
9. The the ActionResult is a success, then the ControllerServlet redirects the HTTP response to the display page for the web application. In the ADK, the display page is typically display.jsp.
10. The `display.jsp` includes the JSP indicated by the `content` parameter; for example, `thanks.jsp`, and displays it to the user.

Design-Time Features

Design-time development has its own features, different from those associated with runtime adapter development. This section describes those features.

Java Server Pages

A design-time GUI is comprised of a set of 11 Java Server Pages. JSPs are simply HTML pages that call Java servlets, which invoke some transaction. To the user, the JSP looks just like any other web page.

The JSPs that comprise a design-time GUI are:

Table 8-2 Design-Time GUI JSPs

Filename	Description
`display.jsp`	The display page, also called the Adapter Home Page; this page contains the HTML necessary to create the look-and-feel of the application view. For instructions on using this page, please refer to "Step 5d: Create display.jsp." This page is supplied by the ADK.
`login.jsp`	The Adapter Design Time Login page.
`confconn.jsp`	The Confirm Connection page; this page provides a form for the user to specify connection parameters for the EIS.
`appvwadmin.jsp`	The Application View Administration page; this page provides a summary of an undeployed application view.
`addevent.jsp`	The Add Event page; this page allows the user to add a new event to the application view.
`addservc.jsp`	The Add Service page; this page allows the user to add a new service to the application view.
`depappvw.jsp`	The Deploy Application View page; this page allows the user to specify deployment properties.

Table 8-2 Design-Time GUI JSPs (Continued)

Filename	Description
`appvwsum.jsp`	The Summary page; this page displays the following information about an application view: ■ Deployed State; that is, whether the application view is deployed or undeployed ■ Connection Criteria ■ Deployment Information (pooling configuration, log level, and security) ■ List of Events ■ List of Services

For a discussion on how to implement these JSPs, please refer to "Step 2: Determining the Screen Flow."

JSP Templates

The design-time framework provides a set of JSP templates for rapidly assembling a web application to define, deploy, and test a new application view for an adapter. A template is an HTML page that is dynamically generated by a Java Servlet based on parameters provided in the HTTP request. Templates are used to minimize the number of custom pages and custom HTML needed for a web application. The templates supplied by the ADK provide three primary features for adapter developers.

- The ADK design-time templates provide most of the HTML forms needed to deploy an application view. In most cases, you will only have to supply three custom forms:
  - One to collect the EIS-specific connection parameters.
  - A second to collects the EIS-specific information needed to add an event.
  - A third to collect the EIS-specific information needed to add a service. In addition, you can supply a custom JSP for browsing a metadata catalog for an EIS.
- The templates also leverage the internationalization and localization features of the Java platform. The content of every page in the web application is stored in a

message bundle. Consequently, the web interface for an adapter can be quickly internationalized.

- The templates centralize look-and-feel into a single location.

Refer to "Developing a Design-Time GUI" for a complete list of JSP templates provided by the ADK.

The ADK Tag Library

The JSP tag library helps to develop user-friendly HTML forms and abstracts complexity from the adapter page developers. Custom tags for form input components allow page developers to seamlessly link to the validation mechanism. Custom tags are provided for the following HTML input tags:

Table 8-3 ADK JSP Tags

Tag	Description
`adk:check box`	Determines if the checkbox form field should be checked when a form is displayed; this tag does not perform validation.
`adk:content`	Provides access to a message in the message bundle.
`adk:date`	Verifies the user's input is a date value that meets a specific format.
`adk:double`	Verifies the user's input is a double value.
`adk:email`	Verifies the user's input is a vaADKlid list of email addresses (one or more).
`adk:float`	Verifies the user's input is a float value.
`adk:identifier`	Verifies the user's input is a valid Java identifier.
`adk:int`	Verifies the user's input is an integer value.
`adk:label`	Displays a label from the message bundle.
`adk:password`	Verifies the user's input in a text field against a Perl 5 regular expression and maskes the input with an asterisk (*).
`adk:submit`	Links the form to the validation mechanism.

Table 8-3 ADK JSP Tags (Continued)

Tag	Description
`adk:text`	Verifies the user's input against a Perl 5 regular expression.
`adk:textarea`	Verifies the user's input into a text area matches a Perl 5 regular expression.
`adk:url`	Verifies the user's input is a valid URL.

JSP Tag Attributes

You can customize the JSP tags by applying the attributes listed in Table 8-4:

Table 8-4 JSP Tag Attributes

Tag	Requires Attributes	Optional Attributes
`adk:int`, `adk:float`, `adk:double`	`name` - field name	`default` - default value on page display `maxlength` - maximum length of value **`size`** - display size `minInclusive` - value supplied by user must be greater than or equal to this value `maxInclusive` - value supplied by user must be less than or equal to this value `minExclusive` - value supplied by user must be strictly greater than this value `maxExclusive` - value supplied by user must be strictly less than this value `required` - (default is false, not required) `attrs` - additional HTML attributes

Table 8-4 JSP Tag Attributes (Continued)

Tag	Requires Attributes	Optional Attributes
adk:date	name - field name	default - default value on page display maxlength - maximum length of value size - display size required - (default is false, field is not required) attrs - additional HTML attributes lenient - should the date formatter be lenient in its parsing? default is false format - the expected format of the user's input, default is "MM/dd/yyyy"
adk:email, adk:url, adk:identifier	name - field name	default - default value on page display maxlength - maximum length of value size - display size required - (default is false, field is not required) attrs - additional HTML attributes
adk:text, adk:password	name - field name	default - default value on page display maxlength - maximum length of value size - display size required - (default is false, field is not required) attrs - additional HTML attributes pattern - a Perl 5 regular expression
adk:textarea	name - field name	default - default value on page display required - (default is false, field is not required) attrs - additional HTML attributes pattern - a Perl 5 regular expression rows - number of rows to display columns - number of columns to display

Note: For more information on tag usage, see `adk.tld` in:

`<WLAI_ROOT>\dev\adk\src\war\WEB-INF\taglibs`

JavaScript Library

The ADK provides JavaScript for opening and closing child windows.

The Application View

The application view represents a business-level interface to the specific functionality in an application. For more information, see “The Application View” in “Introduction to the ADK.”

File Structure

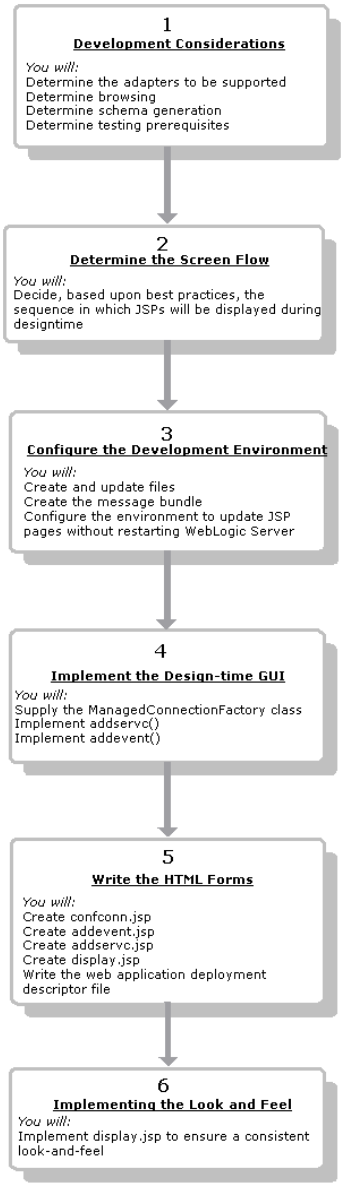
The file structure necessary to build a design-tim GUI adapter is the same as that required for service adapters. See “Step 2a: Set Up the File Structure.” In addition to the structure described there, you should also be aware that:

- The design-time interface for each adapter is a J2EE web application that is bundled as a `.war` file.
- A web application is a bundle of `.jsp`, `.html` and image files.
- The web application descriptor is `WEB-INF/web.xml`. This descriptor instructs the J2EE web container how to deploy and initialize the web application.

The Flow of Events

Figure 8-2 outlines the steps required to develop a design-time GUI.

Figure 8-2 Design-Time GUI Development Flow of Events


1
Development Considerations
You will:
Determine the adapters to be supported
Determine browsing
Determine schema generation
Determine testing prerequisites
2
Determine the Screen Flow
You will:
Decide, based upon best practices, the sequence in which JSPs will be displayed during designtime
3
Configure the Development Environment
You will:
Create and update files
Create the message bundle
Configure the environment to update JSP pages without restarting WebLogic Server
4
Implement the Design-time GUI
You will:
Supply the ManagedConnectionFactory class
Implement addservc()
Implement addevent()
5
Write the HTML Forms
You will:
Create confconn.jsp
Create addevent.jsp
Create addservc.jsp
Create display.jsp
Write the web application deployment descriptor file
6
Implementing the Look and Feel
You will:
Implement display.jsp to ensure a consistent look-and-feel

Step 1: Development Considerations

These are the items you need to consider before commencing with design-time GUI development:

- Determine the adapters to be supported.

  Will this GUI support event adapters? Service adapters? Both?

- Determine browsing.

  The EIS must supply functions to access the event/service catalog. If the EIS does not supply these, the user can't browse the catalogs. If the EIS does supply them, we recommend the following design principle: a call from the design-time UI to get metadata from the EIS is really no different than a call from a runtime component. Both execute functions on the back-end EIS.

  Consequently, you need to leverage your runtime architecture as much as possible to provide the design-time metadata features. You should invoke design time specific functions that use a CCI Interaction object. The sample adapter included with the ADK provides an example/framework of this approach.

  Note: For examples of typical browse sequences, see "How the DBMS Sample Adapter Works" in "The DBMS Sample Adapter" or "How the eMail Sample Adapter Works" in "The eMail Sample Adapter."

- Determine schema generation.

  How will the adapter generate the request/response schema for a service? Will it make a call to the EIS or use some other methodology? Generally, the adapter needs to call the EIS to get metadata about a function or event. The adapter then transforms the EIS metadata into XML schema format. To make this happen, you need to invoke the SOM API. Again, the sample adapter provides instructions for implementing the SOM API. For more information on this API, see "The ADK Tag Library."

- Determine the testing prerequisites.

  Will some sort of service testing be supported? If so, you need to provide:

  - A class that transforms the XML response schema into an HTML form. For an example, see:

`<WLAI_HOME>/dev/dbms/docs/api/com/bea/adapter/dbms/utils/TestFormBuilder.html`

- A JSP named `testform.jsp` that invokes the transformation and displays the HTML form. To see an example of this file, go to `<WLAI_HOME>/dev/dbms/war/`.

Step 2: Determining the Screen Flow

Next, you need to determine the order in which the JSPs will appear when the user displays the application view. This section describes the basic, required screen flow for a successful application view. Note that these are minimum requirements, as you can add more screens to the flow to meet your specific needs.

Screen 1: Logging In

The application view is a secure system, therefore, the user will need to log in before he or she can implement the view. The Application View Management Login page thus must be the first page the user sees.

To use this page, the user supplies a valid username and password. That information is then validated to ensure that the user is a member of the adapter group in the default WebLogic security realm.

Note: The security for the Application View Management web application is specified in the `WEB-INF/web.xml` file, which is shipped in the `wlai.war` file.

Screen 2. Managing Application Views

Once the user successfully logs in, the Application View Management page appears. This page lists the folders that contain the application views, the status of these folders, and any action taken on them. From this page, the user can either view existing application views or add new ones.

- To view an existing application view, the user clicks the appropriate folder and drills down to the desired application view. The user then selects the application view and the Application View Summary page appears (`appvwsum.jsp`; see "Screen 9: Summarizing the Application View").
- To add a new application view, the user clicks Add Application View, which will display the Define New Application View page.

Screen 3: Defining the New Application View

The Define New Application View page (`defappvw.jsp`) allows the user to define a new application view in any folder in which the client is located. To do this, the user needs to provide a description that associates the application view with an adapter. This form provides text boxes for entering the application view name and description and a drop-down list box displaying adapters with which the user can associate the application view.

Once the new adapter is defined, the user selects OK and the Configure Connection page appears.

Screen 4: Configuring the Connection

If the new application view is valid, the user will need to configure the connection. Therefore, once the application view is validated, the next screen in the flow should be the Configure Connection page (`confconn.jsp`). This page provides a form for the user to specify connection parameters for the EIS. Since connection parameters are specific to every EIS, this page is different across all adapters.

When the user submits the connection parameters, the adapter attempts to open a new connection to the EIS using the parameters. If successful, the user is forwarded to the next page, Application View Administration.

Screen 5: Administering the Application View

With a new application view created, the user will need a way of administering it. Therefore, the next screen in the flow should be the Application View Administration page (`appvwadmin.jsp`). This page provides a summary of an undeployed application view. Specifically, it shows the following:

- Connection criteria

  The connection criteria section provides a link that returns the user to the Configure Connection page so that he or she can change connection parameters.

- List of events

  For each event on the application view, the user can do the following:

  - View the XML schema.
  - Remove the event. When the user chooses to remove the event, the system confirms this before removing the event.
  - Provide event properties.

- List of services

  For each service on the application view, the user can do the following:

  - View the request XML schema.
  - View the response XML schema.
  - Remove the service. When the user chooses to remove the service, the system will confirm that choice before removing the event.
  - Provide service properties.

In addition to providing a list of events and a list of services on the application view, the page provides a link to add a new event or service.

Screen 6: Adding an Event

The user will obviously need to add new events to an application view. Therefore, the Application View Administration page contains a link to the Add Event page (`addevent.jsp`). This page allows the user to add a new event to the application view.

The following rules apply to a new event:

- Every event must have a unique name.
  - The event name can only contain a-z, A-Z, 0-9, and underscore (_) and must begin with a letter. Spaces, dots, commas, and so on are not allowed.
  - The length of the name cannot exceed 256 characters.
  - The event name must be unique to the application view. If the user specifies an event name that is not unique, the form will reload with an error message indicating that the event is already defined.
- Optionally, the user can specify a description for the event. This description cannot exceed 2048 (2K) characters.
- In addition to name and description, every event has EIS specific parameters. The collection of EIS-specific parameters define an event type for the adapter.
- Optionally, some adapters provide a mechanism for browsing the event catalog for an EIS.

After adding and saving a new event, the user will be returned to the Application View Administration page.

Screen 7: Adding a Service

As with events, the user will also need to add new services to an application view. Therefore, the Application View Administration page contains a link to the Add Service page (`addservc.jsp`). This page allows the user to add a new service to the application view.

The following rules apply to a new event:

- Every service must have a unique name.
  - The service name can only contain a-z, A-Z, 0-9, and underscore (_) and must begin with a letter. Spaces, dots, commas, and so on are not allowed.
  - The length of the name cannot exceed 256 characters.
  - The service name must be unique to the application view. If the user specifies a service name that is not unique, the form will reload with an error message indicating that the service is already defined.

- Optionally, the user can specify a description for the service. This description cannot exceed 2048 (2K) characters.
- In addition to name and description, every service has EIS specific parameters. The collection of EIS specific parameters define an service type for the adapter.
- Optionally, some adapters provide a mechanism for browsing the service catalog for an EIS.

After adding and saving a new service, the user will be returned to the Application View Administration page.

Screen 8: Deploying an Application View

Once the user adds at least one service or event, he or she can deploy the application view. Deploying an application view makes it available to process events and services. If the user chooses to deploy the application view, he or she will be forwarded to the Deploy Application View page (`depappvw.jsp`).

This screen allows the user to specify deployment properties. The user can specify:

- Connection pooling parameters
  - Minimum pool size: must be greater than or equal to 0
  - Maximum pool size: must be greater than or equal to one.
  - Target fraction of maximum pool size: must be greater than zero and less than one.
  - Allow Pool to Shrink: is the connection pool allowed to shrink?
- Logging level: The user can specify one of four logging levels
  - Log all messages
  - Log informationals, warnings, errors, and audit messages
  - Log warnings, errors, and audit messages
  - Log errors and audit messages
- Security: The user can access a form to apply security restrictions for the application view by clicking on the link that reads Restrict Access. This creates a child window.

Controlling User Access

The user can grant or revoke a user's access privileges by specifying a user or group name in the form provided. Each application view has two types of access: read and write.

- Read access allows the user to execute services and subscribe to events.
- Write access allows the user to deploy/edit/undeploy the application view.

Deploying the Application View

The user deploys the application view by clicking the deploy button. He or she must decide whether or not the application view should be deployed persistently. *Persistent deployment* means that the application view will be redeployed whenever the application server is restarted.

Saving the Application View

The user can save an undeployed application view and return to it later via the Application View Management Console. This process assumes that all deployed application views are saved in the repository. In other words, deploying an unsaved application view will automatically save it.

Screen 9: Summarizing the Application View

Upon successful application view deployment, the user will be forwarded to the Application View Summary page (`appvwsum.jsp`). This page provides the following information about an application view:

- Deployed state: Deployed or Undeployed
  - If the application view is deployed:

    The page will show a link to undeploy the application view. If the user chooses the Undeploy link, a child window will ask the user to confirm his choice to undeploy the application view. If the user confirms, the application view will be undeployed and the summary page will be redisplayed. Undeployed application views are still saved in the repository. This allows the user to edit or remove the application view.

If the adapter supports the testing of events, the Summary page displays a test link for each event. Testing of events is not directly supported by the ADK. Also, if the adapter supports the testing of services, the summary page will display a test link for each service. The ADK demonstrates one possible approach to testing services by providing the `testservc.jsp` and `testrslt.jsp` files. You are free to use these pages to devise your own service testing strategy.

- If the application view is not deployed:

  The page will show a Link to Deploy the application view. If the user chooses the Deploy link, the application view will be deployed and the application view summary page will reload.

  The page will show a link to edit the application view. If the user chooses the Edit link, a child window ask the user to confirm his or her choice to edit the application view. If the user confirms the choice to edit, the Application View Administration page appears.

  The page will show a link to remove the application view. If the user chooses the Remove link, a child window will ask the user to confirm his or her choice to remove the application view from the ADK repository. If the user confirms, the application view will be deleted from the Application Integration repository and the user will be redirected to the adapter main page.

- Connection criteria
- Deployment information (pooling configuration, log level, and security)
- List of events: For each event, there will be a link to view the schema and, if supported, to test the event. The user cannot remove events from this page; they must choose to edit first.
- List of services: For each service, the page will contain a link to view the request schema and the response schema, and, if supported, to test the service. The user cannot remove services from this page; they must undeploy and edit first.

Step 3: Configuring the Development Environment

This step describes the processes you must complete to prepare your computer for design-time GUI development.

Step 3a: Create the Message Bundle

Next, you need to create the message bundle. Any message destined for the end-user should be placed in a message bundle. This bundle is simply a `.properties` text file that contains key=value pairs that allow you to internationalize messages. When a locale and national language are specified at runtime, the contents of the message is interpreted, based upon the key=value pair and the message is presented to the user in the correct language for his or her locale.

For instructions on creating a message bundle, please refer to the JavaSoft tutorial on internationalization at:

`java.sun.com/docs/books/tutorial/i18n/index.html`

Step 3b: Configure the Environment to Update JSPs Without Restarting the WebLogic 6.0 Server

The design-time UI is deployed as a J2EE web application from a `.war` file. A `.war` file is simply a `.jar` file with a web application descriptor in `WEB-INF/web.xml` in the `.jar` file. However, the `.war` file does not allow the J2EE Web container in WebLogic 6.0 to re-compile JSP's on the fly. Consequently, you normally have to restart your WebLogic server just to change a JSP file. Since this goes against the spirit of JSP, the ADK suggests the following workaround to enable you to update JSPs without restarting WebLogic Server 6.0:

1. Construct a valid `.war` file for your adapter's design time UI. For the sample adapter, this is accomplished by using Ant. Listing 8-2 shows the target that produces the J2EE `.war` file:

Listing 8-2 Sample Code Showing Target that Creates a .war File

```
<target name='war' depends='jar'>
<!-- Clean-up existing environment -->
<delete file='${LIB_DIR}/${WAR_FILE}'/>
<delete dir='${SRC_DIR}/war/WEB-INF/lib'/>
<delete dir='${SRC_DIR}/war/WEB-INF/classes'/>
<war warfile='${LIB_DIR}/${WAR_FILE}'
webxml='${SRC_DIR}/war/WEB-INF/web.xml'>
<fileset dir='${PROJECT_DIR}' includes='version_info.xml'/>
<!--
IMPORTANT! Exclude the WEB-INF/web.xml file from the WAR
as it already gets included via the webxml attribute above
-->
<fileset dir='${SRC_DIR}/war' excludes='WEB-INF/web.xml'/>
<!--
IMPORTANT! Include the ADK design time framework into the
adapter's design time Web application.
-->
<fileset dir='${ROOT}/adk/src/war'/>
<!-- Include classes from the adapter that support the design
time UI -->
<classes dir='${SRC_DIR}' includes='sample/web/*.class'/>
<!--
Include all JARs required by the Web application under the
WEB-INF/lib directory of the WAR file
-->
<lib dir='${LIB_DIR}' includes='${JAR_FILE}'/>
```

```
<lib dir='${WLAI_LIB_DIR}'
includes='adk.jar,adk-web.jar,bea.jar,
logtoolkit.jar,webtoolkit.jar,wlai-common.jar,wlai-ejb-client.j
ar,xcci.jar,xmltoolkit.jar'/>

 <lib dir='${RESOURCE_DIR}/log4j' includes='log4j.jar'/>

 <lib dir='${RESOURCE_DIR}/OROMatcher-1.1.0a' includes=
'oromatcher.jar'/>

<lib dir='${RESOURCE_DIR}/xml' includes='xerces_dp1.jar'/>

<lib dir='${RESOURCE_DIR}/xml' includes='xalan.jar'/>

    </war>

<!-- Unjar the WAR into a temp directory; for development -->

<unjar src='${LIB_DIR}/${WAR_FILE}' dest='${LIB_DIR}/
BEA_WLS_SAMPLE_ADK_Web'/>

</target>
```

This Ant target constructs a valid `.war` file for the design time interface in the `<PROJECT_ROOT>/lib` directory, where `<PROJECT_ROOT>` is the location under the Application Integration installation where the developer is constructing the adapter; for example:

`<BEA_HOME\<WLAI_HOME>\dev\sample`

for the sample adapter.

In addition, this target performs an "unjar" operation in the `lib` directory. This extracts the `.war` into a temporary directory. This is the key to having WebLogic recompile JSPs without restarting.

2. Next, load your web application into WebLogic 6.0 server and configure the development environment. Do the following:

   a. To load your web application into WebLogic 6.0 server, you can use the WebLogic console, but we recommend that you edit the `config.xml` file for your domain; for example:

      `<BEA_HOME>\wlserver6.0sp1\config\mydomain\config.xml`.

      Note: If you choose to edit your `config.xml` file, you will need to add an `<application>` element under the domain element:

b. Replace BEA_WLS_SAMPLE_ADK_Web with your adapter's logical name.

c. Replace <WLAI_HOME> with the location of your Application Integration installation; replace <PROJECT_ROOT> with the directory name of your adapter development tree, as shown in Listing 8-3.

Listing 8-3 Sample Code Showing Name of Adapter Development Tree

```
<Application Deployed="true" Name="BEA_WLS_SAMPLE_ADK_Web"
Path="<WLAI_HOME>\dev\<PROJECT_ROOT>\lib">

<WebAppComponent Name="BEA_WLS_SAMPLE_ADK_Web"
ServletReloadCheckSecs="1" Targets="myserver" URI=
"BEA_WLS_SAMPLE_ADK_Web"/>

</Application>
```

Note: If you run GenerateAdapterTemplate, the information in Listing 8-3 will be automatically updated. You can then open <WLAI_HOME>/dev/<CLONE>/src/overview.html and copy and paste it as your config.xml entry.

The key is the URI attribute of the <WebAppComponent> element. Notice that it points to BEA_WLS_SAMPLE_ADK_Web and not BEA_WLS_SAMPLE_ADK_Web.war. This is the temporary directory that you created when you created the .war file. It contains your extracted .war file contents. WebLogic server will watch this directory for JSP changes.

3. To change a JSP, do not change it in the temporary directory; change it from the src/war directory and then rebuild the war target. Remember, when the .war file is created, it is also extracted into the directory WebLogic 6.0 is watching. WebLogic 6.0 will pick up the changes to the specific JSP only. The watch interval used by WebLogic 6.0 is set by the pageCheckSeconds in WEB-INF/weblogic.xml. Listing 8-4 shows how this is done:

Listing 8-4 Sample Code Showing How to Set the Watch Interval

```
<jsp-descriptor>
   <jsp-param>
```

```
      <param-name>compileCommand</param-name>
      <param-value>/jdk130/bin/javac.exe</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>keepgenerated</param-name>
      <param-value>true</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>pageCheckSeconds</param-name>
      <param-value>1</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>verbose</param-name>
      <param-value>true</param-value>
    </jsp-param>
  </jsp-descriptor>
```

This approach also tests whether your `.war` file is being constructed correctly.

4. Finally, you should precompile JSPs when the server starts. This saves you from having to load every page before knowing if they will compile correctly. To enable precompilation you will need to have `weblogic.xml` from the sample adapter and the following element in your `WEB-INF/web.xml` file. Listing 8-5 shows how this is done:

Listing 8-5 Sample Code Showing How to Enable Precompilation of JSPs

```
<context-param>
   <param-name>weblogic.jsp.precompile</param-name>
   <param-value>true</param-value>
</context-param>
```

You can also pre-compile your JSPs using the WebLogic JSP compiler when you build your `.war` target using Ant. This is accomplished by performing the tasks outlined in Listing 8-6 and described here:

- The first task creates the directory where WebLogic 6.0 looks for JSP servlet classes at runtime. There is one caveat to using this approach: you have to specify the target server name. Consequently, it may not suffice as a deployment strategy.
- The second task invokes the JSP compiler `jspc`, provided in WebLogic 6.0. This task pre-compiles all the JSPs for your web application and places them in the `${LIB_DIR}/BEA_WLS_SAMPLE_ADK_Web/WEB-INF/_tmp_war_myserver_myserver_BEA_WLS_SAMPLE_ADK_Web` directory for your web application. Consequently, this allows you to ensure your JSPs will compile every time you build your adapter.

Listing 8-6 Sample Code Showing an Alternate Way to Enable Precompilation of JSPs

```
<mkdir dir='${LIB_DIR}/BEA_WLS_SAMPLE_ADK_Web/
WEB-INF/_tmp_war_myserver_myserver_BEA_WLS_SAMPLE_ADK_Web/jsp_ser
vlet'/>

<!--

This precompiles the JSPs in the Web application during the build.

However, this will only prevent WebLogic from precompiling if the

target server is 'myserver'. If the user is using any other target

server name, the JSP pages will be re-precompiled when the server

starts

-->

<java classname='weblogic.jspc' fork='yes'>

<arg line='-d ${LIB_DIR}/BEA_WLS_SAMPLE_ADK_Web/WEB-INF/
_tmp_war_myserver_myserver_BEA_WLS_SAMPLE_ADK_Web -webapp
${LIB_DIR}/BEA_WLS_SAMPLE_ADK_Web -compileAll -contextPath
BEA_WLS_SAMPLE_ADK_Web -depend -keepgenerated'/>

<classpath refid='CLASSPATH'/>

</java>
```

For more information on precompiling JSPs, see:

`http://e-docs.bea.com/wls/docs60/jsp/reference.html#precompile`

Step 4: Implementing the Design-Time GUI

Implementing the steps described in "Introduction to Design-Time Form Processing" for every form in a web application is a tedious and error prone development process. The design-time framework simplifies this process when you are using a Model-View-Controller paradigm.

To implement the design-time GUI, you need to implement the DesignTimeRequestHandler class. This class accepts user input from a form and performs a design-time action. To implement this class, you must extend the AbstractDesignTimeRequestHandler provided with the ADK; see the Javadoc for this class for a detailed overview of the methods provided by this object.

Extend AbstractDesignTimeRequestHandler

The AbstractDesignTimeRequestHandler provides utility classes for deploying, editing, copying, and removing application views on the Application Integration server. It also provides access to an application view descriptor. The application view descriptor provides the connection parameters, list of events, list of services, log levels, and pool settings for an application view. The parameters are shown on the Application View Summary page.

At a high-level, the AbstractDesignTimeRequestHandler provides an implementation for all actions that are common across adapters. Specifically, these actions are:

- Define the application view.
- Configure the connection.

Note: The ADK provides the method to process connection parameters to obtain a CCI connection but does not supply the `confconn.jsp`. See "Step 5a: Create the confconn.jsp Form" for instructions on creating this form.

- Deploy the application view.
- Provide application view security.
- Edit the application view.
- Undeploy the application view.

Methods to Include

To ensure these actions, you must supply the following methods when you create the concrete implementation of AbstractDesignTimeRequestHandler:

- `initServiceDescriptor();`

  This method adds a service to an application view at designtime (see "Step 4b. Implement initServiceDescriptor()").

- `initEventDescriptor();`

  This method adds an event to an application view at designtime (see "Step 4c. Implement initEventDescriptor()").

You also need to provide in every concrete implementation of `AbstractDesignTimeRequestHandler` the following two methods:

- `protected String getAdapterLogicalName();`

  This method returns the adapter logical name and is used to deploy an application view under an adapter logical name.

- `protected Class getManagedConnectionFactoryClass();`

  This method returns the SPI `ManagedConnectionFactory` implementation class for the adapter.

Step 4a. Supply the ManagedConnectionFactory Class

To supply the ManagedConnectionFactory class, you need to implement the following method:

```
protected Class getManagedConnectionFactoryClass();
```

This method returns the SPI ManagedConnectionFactory implementation class for the adapter. This class is needed by the `AbstractManagedConnectionFactory` when attempting to get a connection to the EIS.

Step 4b. Implement initServiceDescriptor()

For service adapters, you need to implement `initServiceDescriptor()` so that the adapter user can add services at designtime. This method is implemented as shown in Listing 8-7:

Listing 8-7 `initServiceDescriptor()` Implementation

```
protected abstract void initServiceDescriptor(ActionResult result,
                                    IServiceDescriptor sd,
                                    HttpServletRequest request)
     throws Exception
```

This method is invoked by the AbstractDesignTimeRequestHandler's `addservc()` implementation. It is responsible for initializing the EIS-specific information of the `IServiceDescriptor` parameter. The base class implementation of `addservc()` handles the error handling, etc. The `addservc()` method is invoked when the user submits the addservc JSP.

Step 4c. Implement initEventDescriptor()

For event adapters, you will need to implement `initEventDescriptor()` so that the adapter user can add events at designtime. This method is implemented as shown in Listing 8-8:

Listing 8-8 `initEventDescriptor()` Implementation

```
protected abstract void
    initEventDescriptor(ActionResult result,
                        IEventDescriptor ed,
                        HttpServletRequest request)
      throws Exception;
```

This method is invoked by the AbstractDesignTimeRequestHandler's `addevent()` implementation. It is responsible for initializing the EIS-specific information of the `IServiceDescriptor` parameter. The base class implementation of `addevent()` handles such concepts as error handling. The `addevent()` method is invoked when the user submits the addevent JSP. You shouldn't override addevent, as it contains common logic and delegates EIS-specific logic to `initEventDescriptor()`.

Note: When adding properties to a service descriptor, the property names must follow the bean name standard otherwise the service descriptor does not update the InteractionSpec correctly.

Step 5: Write the HTML Forms

The final step to implementing a design-time GUI is to write the various forms that comprise the interface.

- See "Java Server Pages" for a list and description of the necessary forms.
- See "Step 2: Determining the Screen Flow" for the specific details of each form.

The following sections describe how to actually code these forms and include a sample of that code.

Step 5a: Create the confconn.jsp Form

This page provides an HTML form for users to supply connection parameters for the EIS. You are responsible for providing this page with your adapter's design time web application. This form posts to the ControllerServlet with `doAction=confconn`. This implies that the RequestHandler for your design time interface must provide the following method:

```
public ActionResult confconn(HttpServletRequest request) throws
Exception
```

The implementation of this method is responsible for using the supplied connection parameters to create a new instance of the adapter's ManagedConnectionFactory. The ManagedConnectionFactory supplies the CCI ConnectionFactory, which is used to obtain a connection to the EIS. Consequently, the processing of the confconn form submission verifies that the supplied parameters are sufficient for obtaining a valid connection to the EIS.

The confconn form for the sample adapter is shown in Listing 8-9:

Listing 8-9 Coding confconn.jsp

```
<%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>

<form method='POST' action='controller'>

  <table>

    <tr>

      <td><adk:label name='userName' required='true'/></td>

      <td><adk:text name='userName' maxlength='30' size='8'/></td>

    </tr>

    <tr>

      <td><adk:label name='password' required='true'/></td>

      <td><adk:password name='password' maxlength='30'
size='8'/></td>

    </tr>

    <tr>
```

```
      <td colspan='2'><adk:submit name='confconn_submit'
doAction='confconn'/></td>

    </tr>

  </table>

</form>
```

The following paragraphs describe the contents of Listing 8-9.

Including the ADK Tag Library

The line:

```
<%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>
```

instructs the JSP engine to include the ADK tag library. These tags are listed in Table 8-3.

Posting the ControllerServlet

The line:

```
<form method='POST' action='controller'>
```

instructs the form to post to the ControllerServlet. The ControllerServlet is configured in the `web.xml` file for the web application and is responsible for delegating HTTP requests to a method on a RequestHandler. You do not need to provide any code to use the ControllerServlet; however, you must supply the initial parameters, described in Table 8-5:

Table 8-5 ControllerServlet Parameters

Parameter	Description
`MessageBundleBase`	This property specifies the base name for all message bundles supplied with an adapter. The ADK always uses the adapter logical name for its sample adapters. However, you are free to choose your own naming convention for message bundles. Notice that this property is also established in the `ra.xml`.

Table 8-5 ControllerServlet Parameters (Continued)

Parameter	Description
`DisplayPage`	This property specifies the name of the JSP that controls screen flow and look-and-feel. In the sample adapter, this page is `display.jsp`.
`LogConfigFile`	This property specifies the log4j configuration file for the adapter.
`RootLogContext`	This property specifies the root log context. Log context helps categorize log messages according to modules in a program. The ADK uses the adapter logical name for the root log context so that all messages from a specific adapter will be categorized accordingly.
`RequestHandlerClass`	This property provides the fully qualified name of the request handler class for the adapter. In the sample adapter, this value is "sample.web.DesignTimeRequestHandler". See below for details on implementing a DesignTimeRequestHandler.

Displaying the Label for the Form Field

The line:

```
<adk:label name='userName' required='true'/>
```

displays a label for a field on the form. The value that is displayed is retrieved from the message bundle for the user. The "required" attribute indicates if the user must supply this parameter to be successful.

Displaying the Text Field Size

The line:

```
<adk:text name='userName' maxlength='30' size='8'/>
```

sets a text field of size 8 with maximum length (max length) of 30.

Displaying a Submit Button on the Form

The line:

```
<adk:submit name='confconn_submit' doAction='confconn'/>
```

displays a button on the form that allow the adapter user to submit the input. The label on the button will be retrieved from the message bundle using the `confconn_submit` key. When the form data is submitted, the ControllerServlet will locate the `confconn` method on the registered request handler (see the `RequestHandlerClass` property) and pass the request data to it.

Implementing confconn()

The AbstractDesignTimeRequestHandler provides an implementation of the `confconn()` method. This implementation leverages the Java Reflection API to map connection parameters supplied by the user to setter methods on the adapter's ManagedConnectionFactory instance. You only need to supply the concrete class for your adapter's ManagedConnectionFactory by implementing this method:

```
public Class getManagedConnectionFactoryClass()
```

Step 5b: Create the addevent.jsp form

This form allows the user to add a new event to an application view. This form is EIS specific. The `addevent.jsp` form for the sample adapter is shown in Listing 8-10:

Listing 8-10 Sample Code Creating the addevent.jsp Form

```
<%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>

<form method='POST' action='controller'>

  <table>

    <tr>

      <td><adk:label name='eventName' required='true'/></td>

      <td><adk:text name='eventName' maxlength='100'
size='50'/></td>

    </tr>

    <tr>
```

```
      <td colspan='2'><adk:submit name='addevent_submit'
doAction='addevent'/></td>

    </tr>

  </table>

</form>
```

The following paragraphs describe the contents of `addevent.jsp`:

Including the ADK Tag Library

The line:

```
<%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk'%>
```

instructs the JSP engine to include the ADK tag library. These tags are described in Table 8-3.

Posting the ControllerServlet

The line:

```
<form method='POST' action='controller'>
```

instructs the form to post to the ControllerServlet. The ControllerServlet is configured in the `web.xml` file for the web application and is responsible for delegating HTTP requests to a method on a RequestHandler. You do not need to provide any code to use the ControllerServlet; however, you must supply the initial parameters, as described in Table 8-5, "ControllerServlet Parameters."

Displaying the Label for the Form Field

The line:

```
<adk:label name='eventName' required='true'/>
```

displays a label for a field on the form. The value that is displayed is retrieved from the message bundle for the user. The "required" attribute indicates if the user must supply this parameter to be successful.

Displaying the Text Field Size

The line:

```
<adk:text name='eventName' maxlength='100' size='50'/>
```

sets a text field of size 50 with maximum length (max length) of 100.

Displaying a Submit Button on the Form

The line:

```
<adk:submit name='addevent_submit' doAction='addevent'/>
```

displays a button on the form that allow the adapter user to submit the input. The label on the button will be retrieved from the message bundle using the `addevent_submit` key. When the form data is submitted, the "controller" servlet will locate the "addevent" method on the registered request handler (see the RequestHandlerClass property) and pass the request data to it.

Adding Additional Fields

You must also add any additional fields that the user requires for defining an event. See the DBMS or email sample adapters for examples of forms with multiple fields. You can find these files in `<WLAI_HOME>/dev/dbms/src/dbms/` or `<WLAI_HOME>/dev/email/src/email/.`

Step 5c: Create the addservc.jsp form

This form allows the user to add a new service to an application view. This form is EIS-specific. The `addservc.jsp` form for the sample adapter is shown in Listing 8-11:

Listing 8-11 Coding addservc.jsp

```
<%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>
<form method='POST' action='controller'>
  <table>
```

```
    <tr>
      <td><adk:label name='serviceName' required='true'/></td>
      <td><adk:text name='serviceName' maxlength='100'
size='50'/></td>
    </tr>
    <tr>
      <td colspan='2'><adk:submit name='addservc_submit'
doAction='addservc'/></td>
    </tr>
  </table>
</form>
```

Including the ADK Tag Library

The line:

```
<%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>
```

instructs the JSP engine to include the ADK tag library. The ADK tag library supports the user-friendly form validation provided by the ADK. The ADK tag library provides the tags described in Table 8-3.

Posting the ControllerServlet

The line:

```
<form method='POST' action='controller'>
```

instructs the form to post to the ControllerServlet. The ControllerServlet is configured in the `web.xml` file for the web application and is responsible for delegating HTTP requests to a method on a `RequestHandler`. You do not need to provide any code to use the ControllerServlet; however, you must supply the initial parameters as described in Table 8-5, "ControllerServlet Parameters."

Displaying the Label for the Form Field

The line:

```
<adk:label name='servcName' required='true'/>
```

displays a label for the form field. The value that is displayed is retrieved from the message bundle for the user. The "required" attribute indicates if the user must supply this parameter to be successful.

Displaying the Text Field Size

The line:

```
<adk:text name='eventName' maxlength='100' size='50'/>
```

sets a text field of size 50 with maximum length (max length) of 100.

Displaying a Submit Button on the Form

The line:

```
<adk:submit name='addservc_submit' doAction='addservc'/>
```

displays a button on the form that allow the adapter user to submit the input. The label on the button will be retrieved from the message bundle using the `addservc_submit` key. When the form data is submitted, the ControllerServlet will locate the `addservc` method on the registered request handler (see the `RequestHandlerClass` property) and pass the request data to it.

Adding Additional Fields

You must also add any additional fields that the user requires for defining a a service. See the DBMS or email sample adapters for examples of forms with multiple fields. You can find these files in `<WLAI_HOME>/dev/dbms/src/dbms/` or `<WLAI_HOME>/dev/email/src/email/`.

Step 5d: Create display.jsp

This JSP is responsible for establishing the HTML template and including content pages for your web application. It includes the page given by the doAction parameter supplied in the request. The purpose of this JSP is to centralize the HTML that determines look-and-feel into a single JSP for your design time framework.The `display.jsp` page provides the following for the design time web application:

- Establishes the look-and-feel template for all pages.
- Includes other jsp pages based on the `doAction` HTTP request parameter. If the `doAction` HTTP request parameter is not supplied, the `display.jsp` page includes `main.jsp`. You must have `main.jsp`.
- Registers the error page for Java exceptions as `error.jsp` from the ADK.

Since `display.jsp` controls the look-and-feel for all pages, please refer to "Step 7. Implementing the Look-and-Feel" for more information on creating it.

Step 5e: Write the WEB-INF/web.xml Web Application Deployment Descriptor

You will need to create a `WEB-INF/web.xml` web application deployment descriptor for your adapter. When you clone an adapter from the sample adapter by using GenerateAdapterTemplate, a `web.xml` file for that adapter will be automatically generated.

The important components of this file are described in Listing 8-12 through Listing 8-16:

Listing 8-12 web.xml Servlet Components

```
<servlet>
   <servlet-name>controller</servlet-name>
   <servlet-class>com.bea.web.ControllerServlet</servlet-class>
   <init-param>
```

```
     <param-name>MessageBundleBase</param-name>

     <param-value>BEA_WLS_SAMPLE_ADK</param-value>

     <description>The base name for the message bundles for this adapter. The
ControllerServlet uses this name and the user's locale information to determine
which message bundle to use to display the HTML pages.</description>

   </init-param>


   <init-param>

     <param-name>DisplayPage</param-name>

     <param-value>display.jsp</param-value>

    <description>The name of the JSP page that includes content pages and provides
the look-and-feel template. The ControllerServlet redirects to this page to let
it determine what to show the user.</description>

   </init-param>


   <init-param>

     <param-name>LogConfigFile</param-name>

     <param-value>BEA_WLS_SAMPLE_ADK.xml</param-value>

     <description>The name of the sample adapter's LOG4J configuration
file.</description>

   </init-param>


   <init-param>

     <param-name>RootLogContext</param-name>

     <param-value>BEA_WLS_SAMPLE_ADK</param-value>

    <description>The root category for log messages for the sample adapter. All
log messages created by the sample adapter will have a context starting with this
value.</description>

   </init-param>
```

```
   <init-param>
     <param-name>RequestHandlerClass</param-name>
     <param-value>sample.web.DesignTimeRequestHandler</param- value>
     <description>Class that handles design time requests</description>
   </init-param>

   <init-param>
     <param-name>Debug</param-name>
     <param-value>on</param-value>
     <description>Debug setting (on|off, off is default)</description>
   </init-param>
<load-on-startup>1</load-on-startup>
</servlet>
```

This component shown in Listing 8-13 maps the ControllerServlet to the name "controller". This action is important because the ADK JSP forms assume the ControllerServlet is mapped to the logical name "controller".

Listing 8-13 `web.xml` ControllerServlet Mapping Component

```
<servlet-mapping>
   <servlet-name>controller</servlet-name>
   <url-pattern>controller</url-pattern>
</servlet-mapping>
```

This component shown in Listing 8-14 declares the ADK tag library:

Listing 8-14 web.xml ADK Tab Library Component

```
<taglib>
   <taglib-uri>adk</taglib-uri>
   <taglib-location>/WEB-INF/taglibs/adk.tld</taglib-location>
</taglib>
```

This component shown in Listing 8-15 declares the security constraints for the web application. Currently, the user must belong to the adapter group:

Listing 8-15 web.xml Security Constraint Component

```
<Security-constraint>
   <web-resource-collection>
     <web-resource-name>AdapterSecurity</web-resource-name>
     <url-pattern>*.jsp</url-pattern>
   </web-resource-collection>
   <auth-constraint>
     <role-name>adapter</role-name>
   </auth-constraint>
   <user-data-constraint>
     <transport-guarantee>NONE</transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

This component shown in Listing 8-16 declares the login configuration:

Listing 8-16 web.xml Login Configuration Component

```
<login-config>
   <auth-method>FORM</auth-method>
   <realm-name>default</realm-name>
   <form-login-config>
     <form-login-page>/login.jsp</form-login-page>
     <form-error-page>/login.jsp?error</form-error-page>
   </form-login-config>
</login-config>

<security-role>
   <role-name>adapter</role-name>
</security-role>
```

Step 7. Implementing the Look-and-Feel

An important programming practice you should observe when developing a design-time GUI is to implement a consistent look-and-feel across all pages in the application view. The look-and-feel is determined by `display.jsp`. This page is included with the ADK and provides the following for the design time web application:

- Establishes the look-and-feel template for all pages
- Includes other JSPs based on the content HTTP request parameter. If the `content` HTTP request parameter is not supplied, `display.jsp` must include `main.jsp`.
- Registers the error page for Java exceptions as `error.jsp` from the ADK.

To implement a look-and-feel across a set of pages, do the following:

1. Use `display.jsp` from the sample adapter as a starting point. See `sample\src\war\WEB-INF\web.xml` for an example.
2. Using HTML, alter the look-and-feel markup in this page to reflect your own look-and-feel or company identity standards.
3. Somewhere in your HTML markup, be sure to include:

   ```
   <%pageContext.include(sbPage.toString());%>
   ```

   This code is a custom JSP tag used to include other pages. This tag uses the JSP scriptlet "`sbPage.toString()`" to include an HTML or JSP into the display page. `sbPage.toString()` evaluates to the value for the HTTP request parameter content at runtime.

APPENDIX

A Adapter Setup Worksheet

Use the worksheet beginning on the following page to help you identify and collect critical information about the adapter you are developing. The answers to the questions posed on the worksheet will help you conceptualize the adapter you are building before you actually began to code. They will help you define such components as the adapter logical name and the Java package base name and help you determine the locales for which you need to localize message bundles. If you are using the GenerateAdapterTemplate utility, the answers you provide on this worksheet are essential to its success.

Adapter Setup Worksheet

Before you begin developing an adapter, answer as many of the following questions as you can. Questions preceded by an asterisk (*) are required to use the GenerateAdapterTemplate utility.

1. *What is the name of the EIS for which you are developing an adapter?
2. *What is the version of the EIS?
3. *What is the type of EIS; for example, DBMS, ERP, etc.?
4. *Who is the vendor name for this adapter?
5. *What is the version number for this adapter?
6. *What is the adapter logical name?
7. Does the adapter need to invoke functionality within the EIS?

   If so, then your adapter needs to support services.
8. What mechanism/API is provided by the EIS to allow an external program to invoke functionality provided by the EIS?
9. What information is needed to create a session/connection to the EIS for this mechanism?
10. What information is needed to determine which function(s) will be invoked in the EIS for a given service?
11. Does the EIS allow you to query it for input and output requirements for a given function?

    If so, what information is needed to determine the input requirements for the service?
12. For all the input requirements, which ones are static across all requests? Your adapter should encode static information into an InteractionSpec object.
13. For all the input requirements, which ones are dynamic per request? Your adapter should provide an XML schema that describes the input parameters required by this service per request.

14. What information is needed to determine the output requirements for the service?
15. Does the EIS provide a mechanism to browse a catalog of functions your adapter can invoke? If so, your adapter should support browsing of services.
16. Does the adapter need to receive notifications of changes that occur inside the EIS? If so, then your adapter needs to support events.
17. What mechanism/API is provided by the EIS to allow an external program to receive notification of events in the EIS? The answer of this question will help determine if a pull or a push mechanism is developed.
18. Does the EIS provide a way to determine which events your adapter can support?
19. Does the EIS provide a way to query for metadata for a given event?
20. What locales (language/country) does your adapter need to support?

A Adapter Setup Worksheet

APPENDIX

B The DBMS Sample Adapter

This section contains information on the following subjects:

- Introduction to the DBMS Sample Adapter
- How the DBMS Sample Adapter Works
- How the DBMS Sample Adapter Was Developed
- How the DBMS Adapter Design-Time GUI was Developed

Introduction to the DBMS Sample Adapter

The DBMS adapter is a J2EE-compliant adapter built with the BEA WebLogic Application Integration Adapter Development Kit. It provides a concrete example for adapter providers of how one might use the ADK to construct an adapter. A relational database was used as the EIS of an adapter because it allows adapter providers to focus on the adapter/ADK specifics, rather than become bogged-down in understanding a particular proprietary EIS.

The DBMS sample adapter gives you (developers and business analysts) a concrete example of an adapter, including a JSP-based GUI, to help you understand the possibilities that are at your disposal using the ADK to build adapters. If you are a business analyst, you might enjoy running through the interface to get a better understanding of an “application view”, “service”, and “event” as shown in “How the DBMS Sample Adapter Works.”

If you are an adapter developer, you will also want to review "How the DBMS Sample Adapter Was Developed" and "How the DBMS Adapter Design-Time GUI was Developed," the code, and Javadoc to gain insight into how you can extend and use the classes of the ADK to build a JCA-compliant adapter.

The DBMS adapter satisfies the following requirements:

- Provides a GUI that allows end-users to connect to a Cloudscape, Oracle, or SQLServer database.
- Uses the classes and tools of the ADK.
- Allows users to create application views with events and services.
- Allow users to test events and services.
- Provides a GUI that enables users to browse the catalogs, schemas, tables, and columns of the underlying database from the GUI.
- Supports the creation of services that perform selects, inserts, deletes, and updates against the database (EIS).

How the DBMS Sample Adapter Works

This section provides you with an opportunity to see how the DBMS adapter works before you start developing one of your own. If you are a business analyst, you might enjoy running through the interface to get a feel for how the adapter works. The example in this section shows how to create a service that inserts a customer in the underlying database, and then demonstrates how an event is generated to notify others that this action has taken place.

This section contains information on the following subjects:

- Before You Begin
- Accessing the DBMS Sample Adapter
- A Tour of the DBMS Sample Adapter

Before You Begin

Make sure the following tasks have been performed before you try to access the DBMS sample adapter:

- Install the BEA WebLogic Application Integration. For more information, see the *BEA WebLogic Application Integration Installation and Setup Guide.*
- Set up the ADK Ant-Based Make Process.
- Ensure that the DBMS sample adapter has been deployed so that the design-time GUI is accessible. For more information, see the *BEA WebLogic Application Integration Installation and Configuration Guide*.

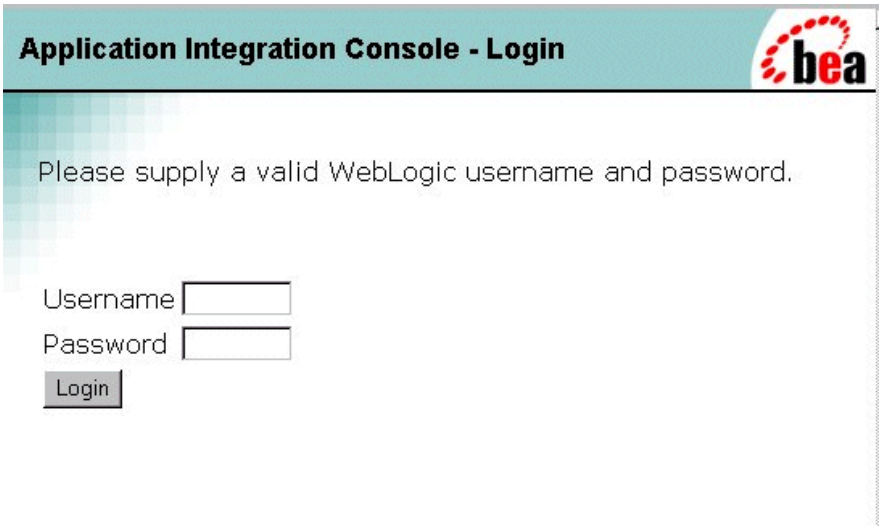
Accessing the DBMS Sample Adapter

To access the DBMS adapter:

1. Open a new browser window.
2. Open the URL for your system's Application View Management Console.

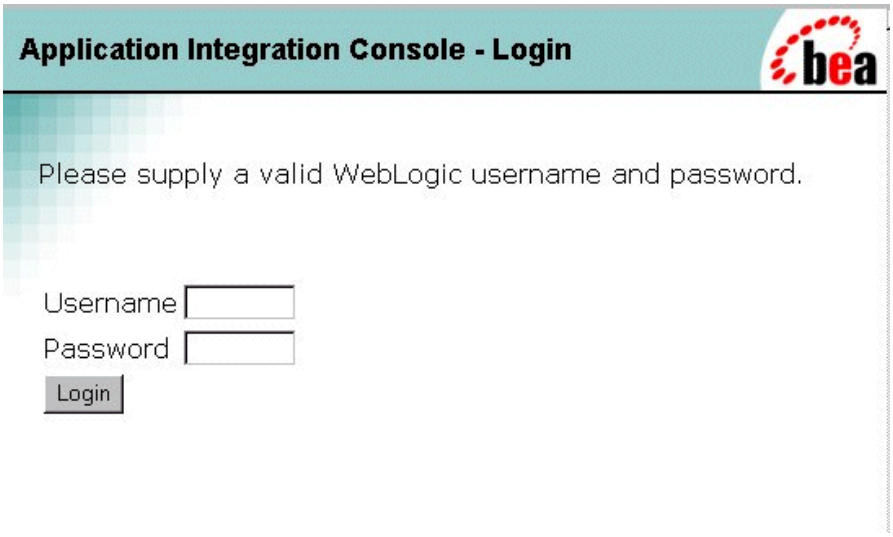
   `http://<HOSTNAME>:7601/wlai`

   The Application Integration Console - Login page displays.

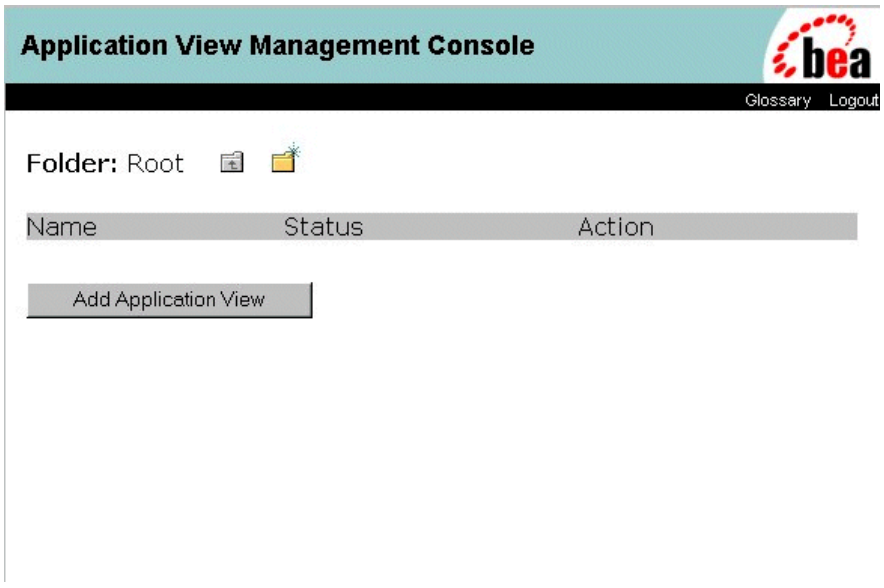


A Tour of the DBMS Sample Adapter

This section provides you with a short tour through the DBMS Sample adapter. Before you begin, you need to open the DBMS adapter Login page on your browser. For information about accessing the DBMS adapter, see "Accessing the DBMS Sample Adapter."

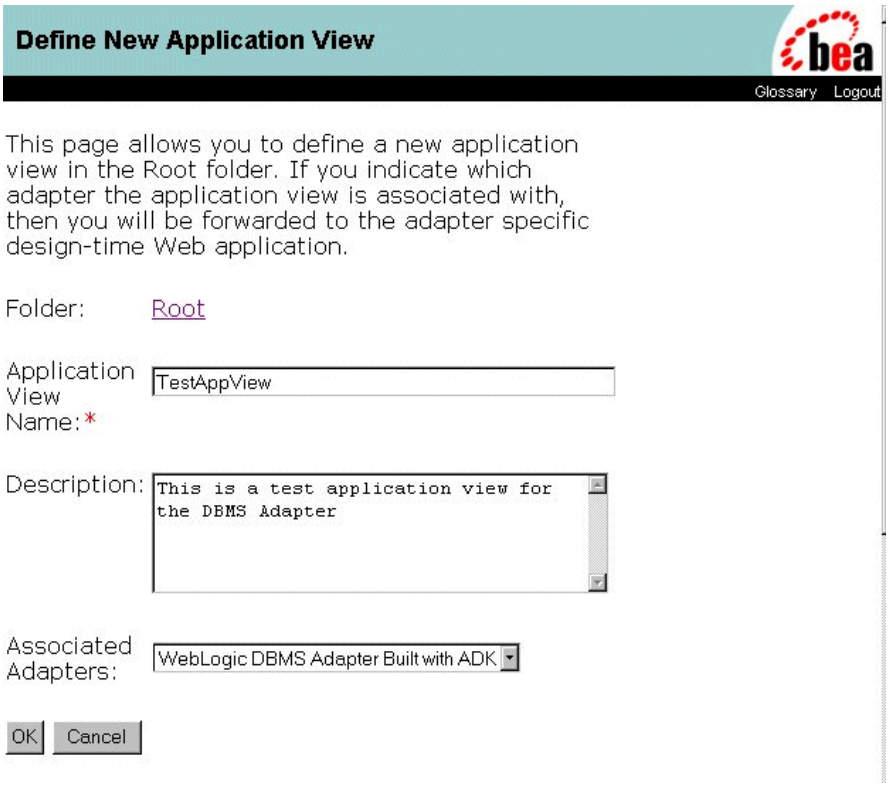


1. To log on to the Application View Management Console, enter your WebLogic username and password, then click Login. The Application View Management Console displays.



2. Click Add Application View. The Define New Application View page displays. When you create the application view, you provide a description that associates the application view with the DBMS adapter.

Note: For detailed information about application views, see "Introduction to Using Application Integration." For detailed information about defining application views, see "Defining Application Views" in the *BEA WebLogic Application Integration User Guide*.



3. To define an application view:

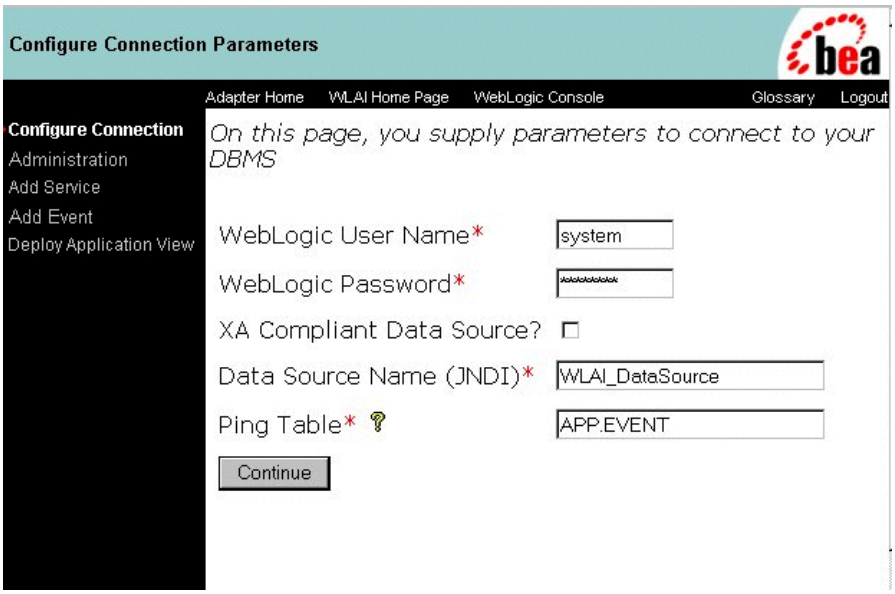
   a. In the Application View Name field, enter TestAppView.

Note: The name should describe the set of functions performed by this application. Each application view name must be unique to its adapter. Valid characters are anything except '.', '#', '\', '+', '&', ',', ''', '"", and a space.

b. In the Description field, enter a brief description of the application view.

c. From the Associated Adapters list, choose the DBMS adapter to use to create this application view.

d. Click OK. The Configure Connection Parameters page displays.



4. At the Configure Connection Parameters page, you define the network-related information necessary for the application view to interact with the target EIS. You need to enter this information only once per application view:

a. Enter your WebLogic User Name and WebLogic Password.

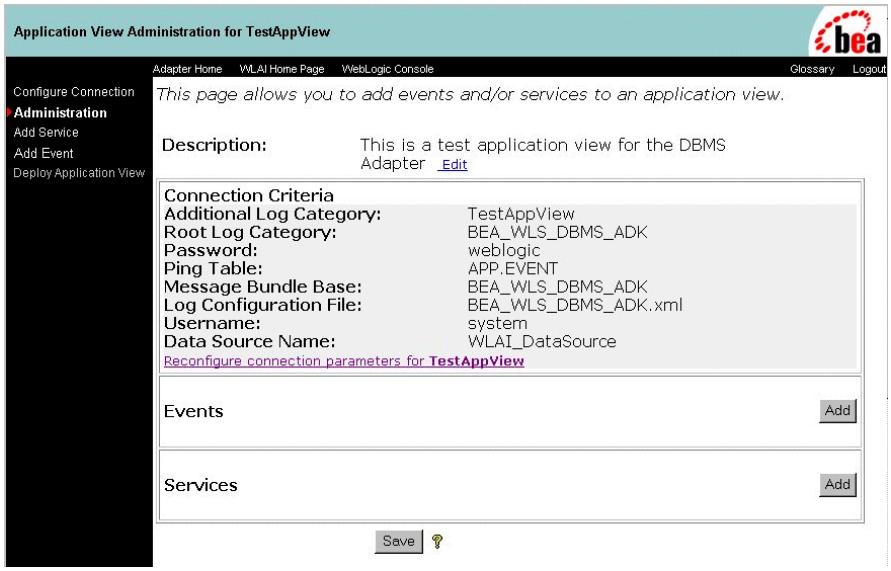
b. Make sure the XA Compliant Data Source? box is unchecked.

c. In the Data Source Name (JNDI) field, enter WLAI_DataSource.

d. In the Ping Table field, enter APP.EVENT.

e. Click Continue. The Application View Administration page displays.

Note: The Administration page summarizes the connection criteria, and once events and services are defined, you can view the schemas and summaries and also delete an event or service from this page.

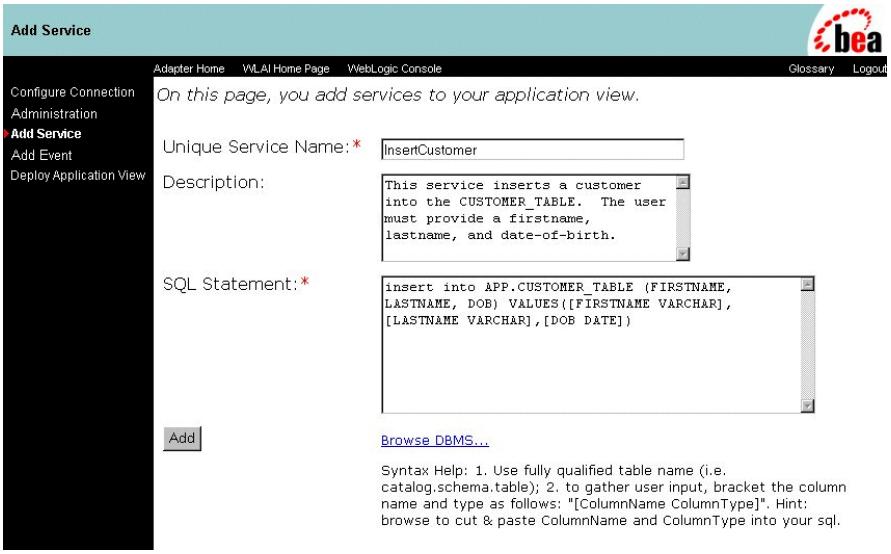


5. Now that you have created an application view, you are ready to add a service to it. To add the service you must supply a name for the service, provide a description and enter the SQL statement.

   You can use the browse link to browse the DBMS adapter database schemas and tables and specify the database table CUSTOMER_TABLE.

   To add a service:

   a. In the Administration page, click Add in the Services field. The Add Service page displays.



b. In the Unique Service Name field, enter InsertCustomer.

c. In the Description field, enter a description of the service.

d. Click Browse DBMS to view the table and column structure of the database. If you are writing a complex query, you may leave the Browse window open in order to cut and paste table or column names into your query.

e. In the DBMS Schemas for Catalog page, click APP.

f. In the DBMS Table Types page, click TABLE.

g. In the Tables list for APP page, click CUSTOMER_TABLE. The Browse window now displays the column names and column types.

Browse DBMS

DBMS Columns For Table: CUSTOMER_TABLE

ColumnName:	ColumnType:	ColumnSize:
FIRSTNAME	VARCHAR	32
LASTNAME	VARCHAR	32
MIDDLENAME	VARCHAR	32
DOB	DATE	10
ADDRESS1	VARCHAR	32
ADDRESS2	VARCHAR	32
ADDRESS3	VARCHAR	32
POSTALCODE	VARCHAR	11
CITY	VARCHAR	32
STATE	VARCHAR	32
COUNTRY	VARCHAR	32
PHONE	VARCHAR	15
FAX	VARCHAR	15
EMAIL	VARCHAR	64

h. Click Close to close to close the window to return to the Add Service Page.

Note: This window is included in the tour to introduce you to the functionality, and it is not necessary to select any text for this exercise.

i. In the Service Page, add the following information into the SQL Statement field:

```
Insert into APP.CUSTOMER_TABLE (FIRSTNAME, LASTNAME, DOB)
VALUES ([FIRSTNAME VARCHAR], [LASTNAME VARCHAR], [DOB
DATE])
```

j. Click Add. The Administration page displays.

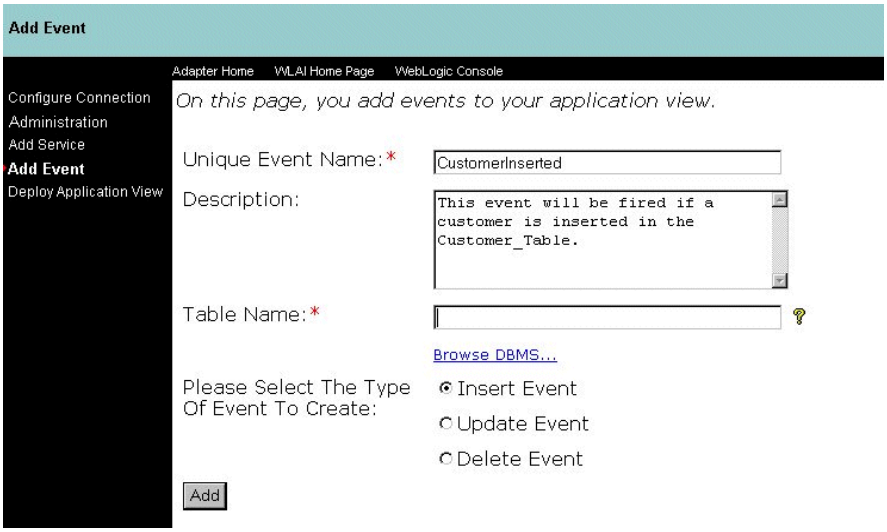
Note: For additional information about adding services, see “Adding Services to an Application View” in the *BEA WebLogic Application Integration User Guide.*

6. Add an event to your application view. In order to add an event, you must provide a unique event name and a description. Then you must specify the database table upon which a trigger should be added for the event. You must also specify if it is an insert, update or delete event.

   You can use the Browse DBMS link to browse the DBMS database schemas and tables and to specify the database table. Then you can automatically populate the field with the selected table name.

   To add an event:

   a. In the Administration page, click Add in the Event field. The Add Event page displays.

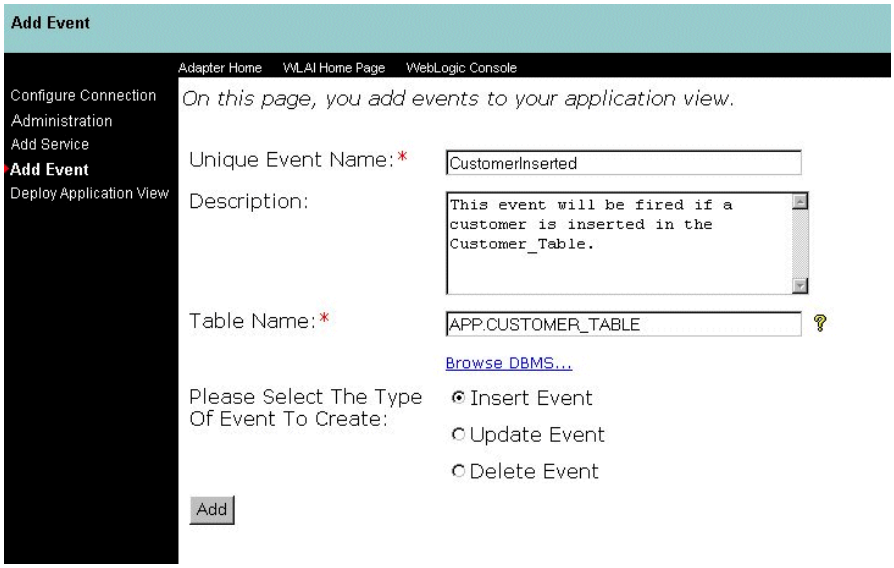


   b. In the Unique Event Name field, enter CustomerInserted.

   c. In the Description field, enter a description of the event.

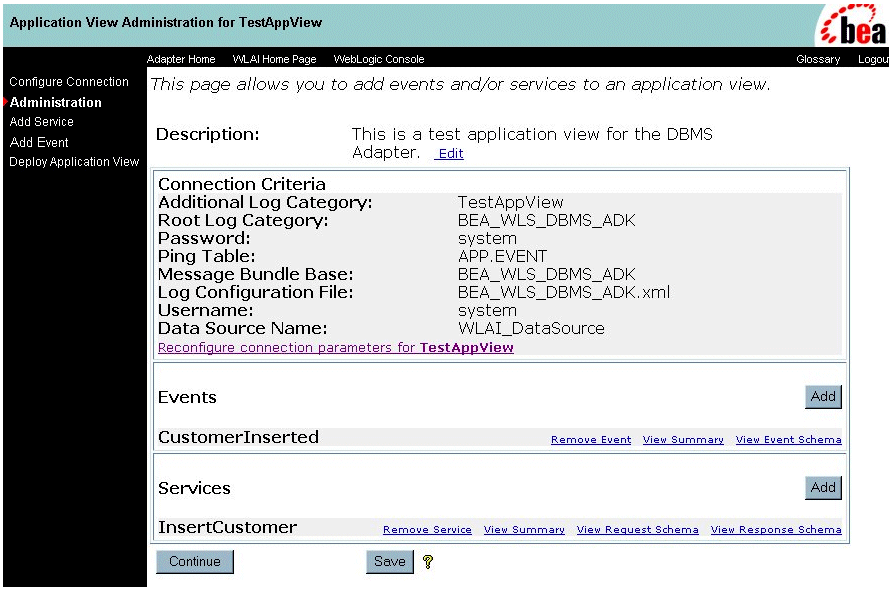
   d. Click the Browse DBMS link to view the table and column structure of the database.

e. Select the CUSTOMER TABLE radio button, and click Fill table name with selected table.



f. Select the Insert Event radio button.

g. When finished, click Add. The Application View Administration page displays.

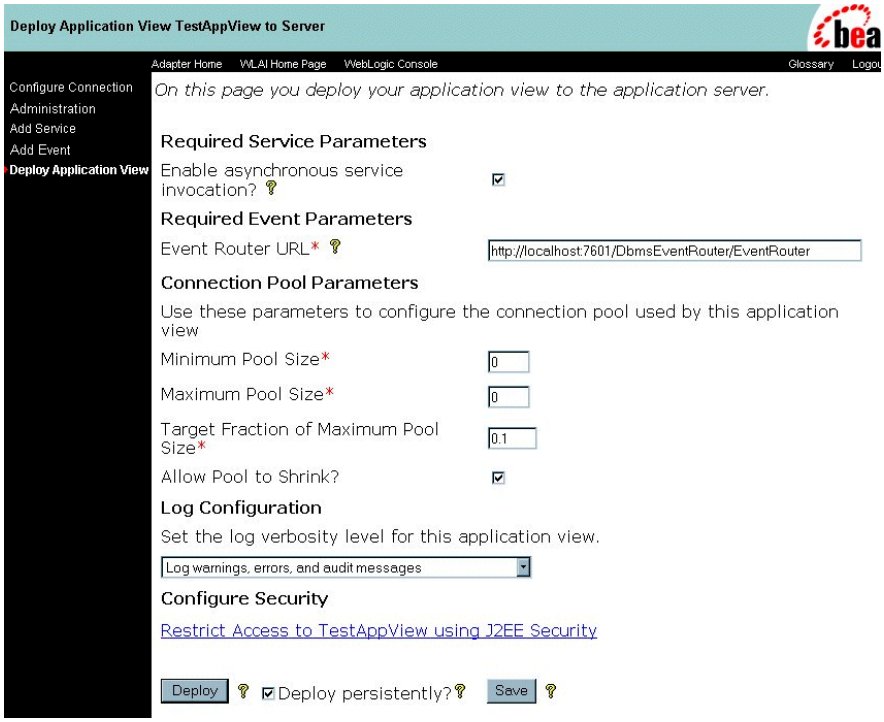


7. Prepare to deploy the application view. The Application View Administration page provides you with a single location for confirming the content of your application view before you save it or deploy it. In this page, you can view the following:

   - Confirm or edit the description of the application view.
   - Confirm or reconfigure Connection Criteria for the application view.
   - Delete services and events.
   - Save the application view so you can return to it later or deploy the application view to the server.

   After verifying the application view parameters, click Continue. The Deploy Application View to Server page displays.

8. Deploy the Application View. In order to deploy the application view, you must provide several parameters such as enabling asynchronous service invocation, providing the event router URL, and changing the connection pool parameters, among other parameters.



To deploy the application view:

a. Make sure the Enable Asynchronous Service Invocation check box is checked.

b. In the Event Router URL field, enter
`http://localhost:7601/DbmsEventRouter/EventRouter`

c. For the Connection Pool Parameters, accept the default values:

   Minimum Pool Size - 1

   Maximum Pool Size - 10

   Target Fraction of Maximum Pool Size - 0.7

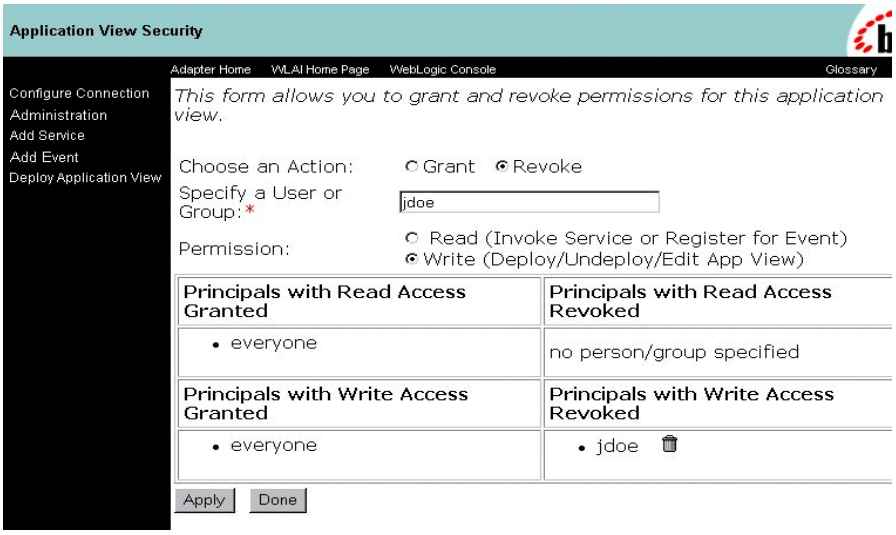
   Allow Pool to Shrink - checked

d. In the Log Configuration field, select Log warnings, errors, and audit messages.

e. Make sure the Deploy persistently? box is checked.

f. Click the Restrict Access link. The Application View Security page displays.

9. Set permissions for the application view. You can grant or revoke read and write access for a user or a group.



To set permissions for the application view:

a. For Choose an Action, select the Revoke radio button.

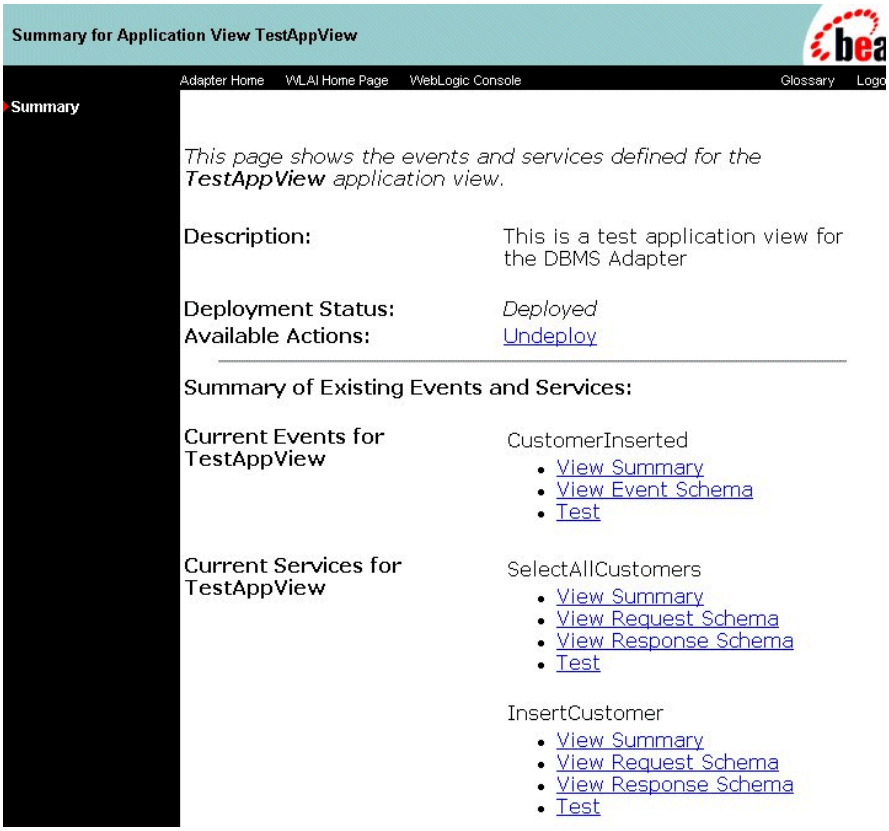
b. In the Specify a User or Group, enter Jdoe.

c. For Permission: select the Write (Deploy/Undeploy/Edit App View) radio button.

d. Click Done. The Deploy Application View Page displays.

e. Click Deploy.

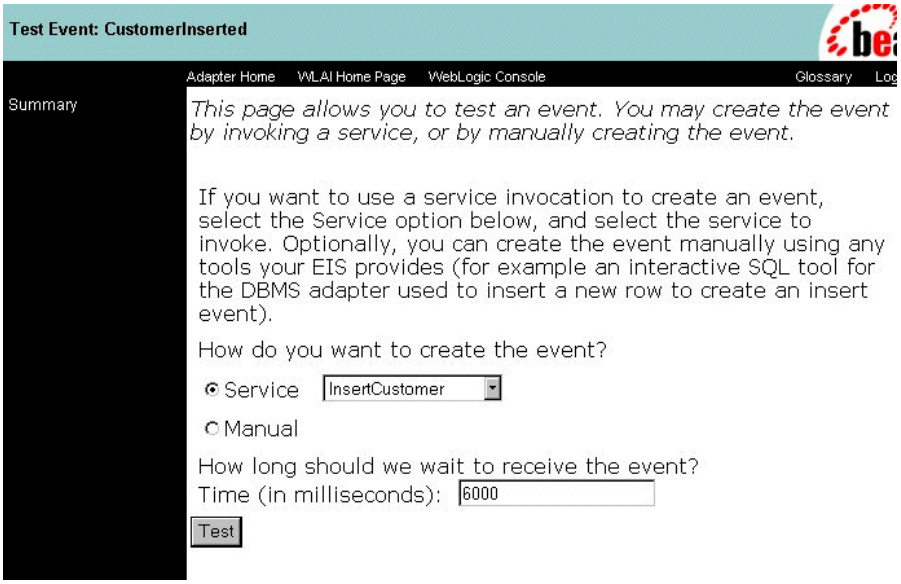
10. Once the application view is deployed, the summary page displays all relevant information about the deployed application view. Use the Summary page to view schemas, event and service summaries, test services and events and undeploy the application view.



11. Test an event. To ensure that the application view is working correctly, you can test the events and services in the application view. You can test an event by invoking a service or by manually creating the event. The user can also specify how long the application should wait to receive the event.

    To test an event using a service:

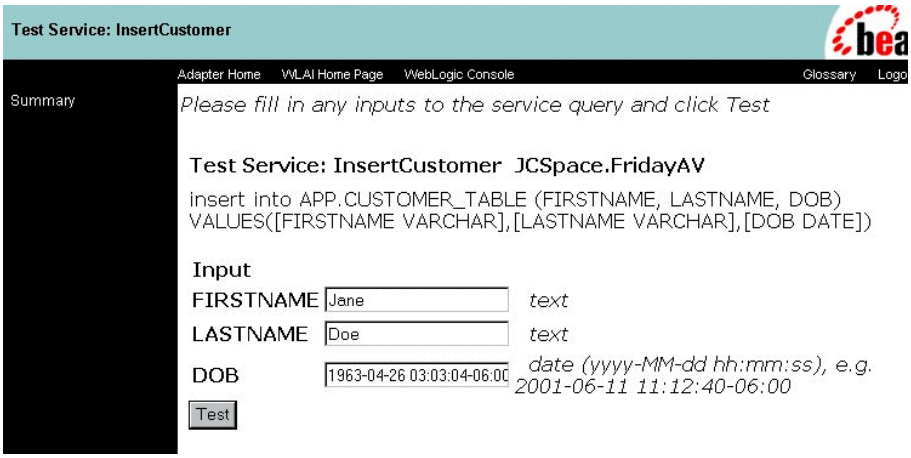
    a. From the Summary page, click the Test link for CustomerInserted. The Test Event page displays.



b. In the Test Event page, select the Service radio button, and select InsertCustomer in the Service menu.

c. In the How long should we wait to receive the event? field, enter 6000.

d. Click Test. The Test Service page displays.

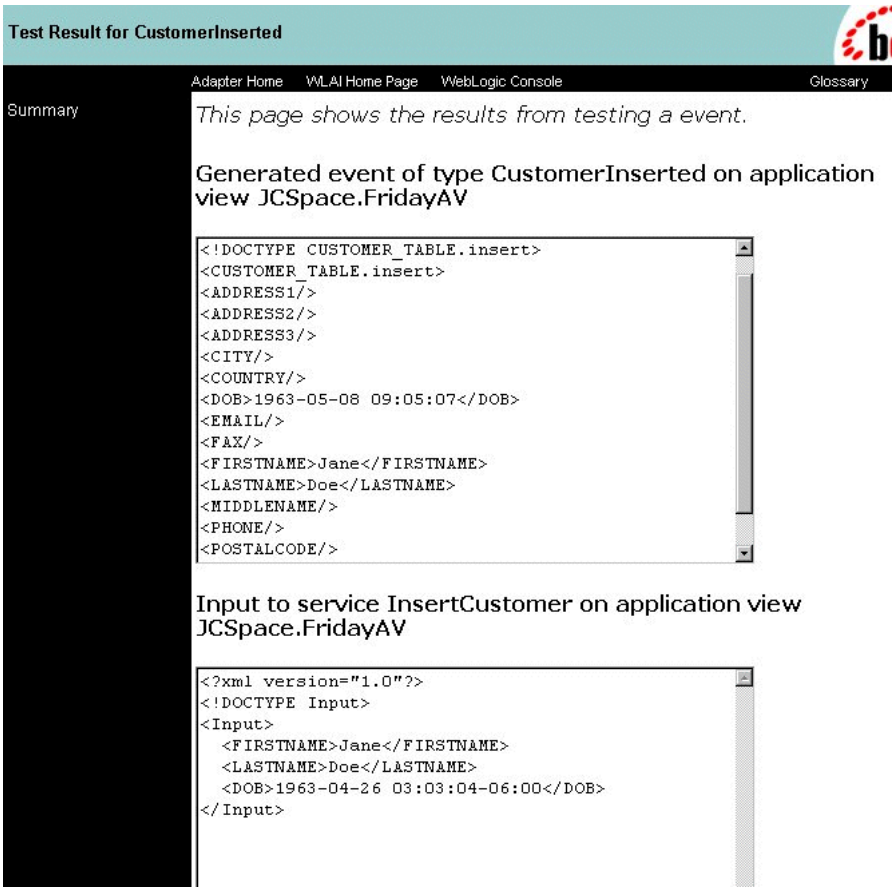


e. In the FIRSTNAME field, enter a first name.

f. In the LASTNAME field, enter a last name.

g. In the DOB field, enter a date of birth in the correct format.

h. Click Test. The Test Result page displays to show the contents of the XML documents representing the event you generated and the response generated by the application view.



How the DBMS Sample Adapter Was Developed

This section describes each interface used to develop the DBMS adapter. The ADK provides many of the necessary implementations required by a Java Connector Architecture-compliant adapter; however, since some interfaces cannot be fully implemented until the EIS and its environment are defined, the DBMS adapter was created to illustrate the detail-specific or concrete implementation of the abstract classes provided in the ADK.

The process of creating the DBMS adapter is comprised of the following steps:

- Development Reference Documentation
- Step 1: Development Considerations
- Step 2: Implementing the Server Provider Interface Package
- Step 3: Implementing the Common Client Interface Package
- Step 4: Implementing the Event Package
- Step 5: Deploying the Adapter

Development Reference Documentation

You can review the Javadoc and code for the methods defined in the steps that follow in this section to see how the implementations provided by the ADK were leveraged. You can find the Javadoc for this implementation in:

`<WLAI_HOME>\dev\dbms\docs\api\index.html`

You can find the code listing for this package in:

`<WLAI_HOME>\dev\dbms\src\com\bea\adapter\dbms\spi`

Note: `<WLAI_HOME>` is the drive or home directory where BEA WebLogic Application Integration is installed.

Step 1: Development Considerations

The "Adapter Setup Worksheet" is available to help adapter developers identify and collect critical information about an adapter they are developing before they begin coding. For the DBMS sample adapter, the worksheet questions are answered as follows:

Note: Questions preceded by an asterisk (*) are required to use the GenerateAdapterTemplate utility.

1. **What is the name of the EIS for which you are developing an adapter?*

   Cloudscape, SQLServer, Oracle databases.

2. **What is the version of the EIS?*

   Cloudscape, MSSQLServer 7.0 Oracle 8.1.6

3. **What is the type of EIS; for example, DBMS, ERP, etc.?*

   DBMS

4. **What is the vendor name for this adapter?*

   BEA

5. **What is the version number of this adapter?*

   None - Sample Only

6. **What is the adapter logical name?*

   BEA_WLS_DBMS_ADK

7. *Does the adapter need to invoke functionality within the EIS?*

   Yes

   If so, then your adapter needs to support services.

   Yes

8. *What mechanism/API is provided by the EIS to allow an external program to invoke functionality provided by the EIS?*

   JDBC

9. *What information is needed to create a session/connection to the EIS for this mechanism*?

   Database URL, driver class, user name, password

10. *What information is needed to determine which function(s) will be invoked in the EIS for a given service?*

    Function name, executeUpdate, executeQuery

11. *Does the EIS allow you to query it for input and output requirements for a given function?*

    Yes, you can browse data structures.

    If so, what information is needed to determine the input requirements for the service?

    SQL

12. *For all the input requirements, which ones are static across all requests? Your adapter should encode static information into an InteractionSpec object.*

    SQL

13. *For all the input requirements, which ones are dynamic per request? Your adapter should provide an XML schema that describes the input parameters required by this service per request.*

    The input requirements would change depending on the SQL expression for the service

14. *What information is needed to determine the output requirements for the service?*

    n/a

15. *Does the EIS provide a mechanism to browse a catalog of functions your adapter can invoke? If so, your adapter should support browsing of services.*

    Yes

16. *Does the adapter need to receive notifications of changes that occur inside the EIS? If so, then your adapter needs to support events.*

    Yes

17. *What mechanism/API is provided by the EIS to allow an external program to receive notification of events in the EIS? The answer of this question will help determine if a pull or a push mechanism is developed.*

    None. The DBMS adapter was built on the BEA WebLogic Application Integration event generator using a pull mechanism.

18. *Does the EIS provide a way to determine which events your adapter can support?*

    Yes

19. *Does the EIS provide a way to query for metadata for a given event?*

    Yes

20. *What locales (language/country) does your adapter need to support?*

    Multiple

Step 2: Implementing the Server Provider Interface Package

To implement the DBMS adapter Server Provider Interface (SPI) and meet the J2EE-compliant SPI requirements, the classes in the ADK were extended to create the following concrete classes:

Table 8-6 SPI Class Extensions

This concrete class...	Extends this ADK class...
`ManagedConnectionFactoryImpl`	`AbstractManagedConnectionFactory`
`ManagedConnectionImpl`	`AbstractManagedConnection`
`ConnectionMetaDataImpl`	`AbstractConnectionMetaData`
`LocalTransactionImpl`	`AbstractLocalTransaction`

These classes provide connectivity to an EIS and establish a framework for event listening and request transmission, establish transaction demarcation, and allow management of a selected EIS.

ManagedConnectionFactoryImpl

The first step in implementing an SPI for the DBMS adapter was to implement the ManagedConnectionFactory interface. A ManagedConnectionFactory supports connection pooling by providing methods for matching and creating a ManagedConnection instance.

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractManagedConnectionFactory`, an implementation of the Java Connector Architecture interface `javax.resource.spi.ManagedConnectionFactory`. The DBMS adapter extends this class in `com.bea.adapter.dbms.spi.ManagedConnectionFactoryImpl`. Listing 8-17 shows the derivation tree for `ManagedConnectionFactoryImpl`.

Listing 8-17 com.bea.adapter.dbms.spi.ManagedConnectionFactoryImpl

```
javax.resource.spi.ManagedConnectionFactory
  |
  |-->com.bea.adapter.spi.AbstractManagedConnectionFactory
        |
        |-->com.bea.adapter.dbms.spi.ManagedConnectionFactoryImpl
```

Developers' Comments

The ManagedConnectionFactory is the central class of the Java Connector Architecture's SPI package. The ADK's `AbstractManagedConnectionFactory` provides much of the required implementation for the methods declared in Sun's interface. To extend the ADK's `AbstractManagedConnectionFactory` for the DBMS adapter, the key `createConnectionFactory()` and `createManagedConnection()` methods were implemented. Overrides for `equals()`, `hashcode()`, `checkState()` were also written to provide specific behaviors for the databases supported by the DBMS adapter.

There are private attributes about which the superclass has no knowledge. When creating your adapters, you must provide setter/getter methods for these attributes. The abstract `createConnectionFactory()` method is implemented to provide an EIS-specific ConnectionFactory using the input parameters.

Additionally, `createManagedConnection()` is the main factory method of the class. It checks to see if the adapter is configured properly before doing anything else. It then implements methods of the superclass to get a connection and a password credential object. It then attempts to open a physical database connection; if this succeeds, it instantiates and returns a `ManagedConnectionImpl` (the DBMS adapter implementation of `ManagedConnection`), which is given the physical connection.

Note: The `createManagedConnection()` method uses our member `m_bIsXaCompliantDataSource` boolean to conditionally open either an XA or a non-XA compliant datasource. The methods it invokes to do this are private and are interesting to review.

Other methods are getter/setter methods for member attributes.

ManagedConnectionImpl

A ManagedConnection instance represents a physical connection to the underlying EIS in a managed environment. ManagedConnection objects are pooled by the application server. For more information, read about how the ADK implements the AbstractManagedConnection instance in "ManagedConnection."

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractManagedConnection`, an implementation of the J2EE interface `javax.resource.spi.ManagedConnection`. The DBMS adapter extends this class in `com.bea.adapter.dbms.spi.ManagedConnectionImpl`. Listing 8-18 shows the derivation tree for `ManagedConnectionImpl`.

Listing 8-18 com.bea.adapter.dbms.spi.ManagedConnectionImpl

```
javax.resource.spi.ManagedConnection
  |
  |-->com.bea.adapter.spi.AbstractManagedConnection
        |
        |-->com.bea.adapter.dbms.spi.ManagedConnectionImpl
```

Developers' Comments

This class is well documented in the Javadoc comments since the ManagedConnection is a crucial part of the Java Connector Architecture SPI specification. You should focus on our implementation of the following methods:

- `java.lang.object.getConnection(javax.security.auth.Subject subject, javax.resource.spi.ConnectionRequestInfo connectionRequestInfo)`
- `protected void destroyPhysicalConnection(java.lang.Object objPhysicalConnection)`
- `protected void destroyConnectionHandle(java.lang.Object objHandle)`
- `boolean compareCredentials(javax.security.auth.Subject subject, javax.resource.spi.ConnectionRequestInfo info)`

The `ping()` method is used to check whether the physical database connection (not our cci.`Connection`) is still valid. If an exception occurs, `ping()` is very specific about checking the type so that a connection is not needlessly destroyed.

Other methods are EIS specific or are simply required setter/getters.

ConnectionMetaDataImpl

The ManagedConnectionMetaData interface provides information about the underlying EIS instance associated with a ManagedConnection instance. An application server uses this information to get runtime information about a connected EIS instance. For more information, read about how the ADK implements the AbstractConnectionMetaData instance in "ManagedConnection."

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractConnectionMetaData`, an implementation of the J2EE interface `javax.resource.spi.ManagedConnectionMetaData`. The DBMS adapter extends this class in `com.bea.adapter.dbms.spi.ConnectionMetaDataImpl`. Listing 8-19 shows the derivation tree for `ConnectionMetaDataImpl`.

Listing 8-19 com.bea.adapter.dbms.spi.ConnectionMetaDataImpl

```
javax.resource.spi.ManagedConnectionMetaData
  |
  |-->com.bea.adapter.spi.AbstractConnectionMetaData
       |
       |-->com.bea.adapter.dbms.spi.ConnectionMetaDataImpl
```

Developers' Comments

The ADK's `AbstractConnectionMetaData` implements the following:

- `javax.resource.cci.ConnectionMetaData`
- `javax.resource.spi.ManagedConnectionMetaData`

This implementation of the ConnectionMetaData class uses a DatabaseMetaData object. Since the ADK's abstract implementation was used, you must provide EIS-specific knowledge when implementing the abstract methods in this class.

LocalTransactionImpl

The LocalTransaction interface provides support for transactions that are managed internal to an EIS resource manager and do not require an external transaction manager. For more information, read about how the ADK implements the AbstractLocalTransaction instance in "LocalTransaction."

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractLocalTransaction`, an implementation of the J2EE interface `javax.resource.spi.LocalTransaction`. The DBMS adapter extends this class in `com.bea.adapter.dbms.spi.LocalTransactionImpl`. Listing 8-20 shows the derivation tree for `LocalTransactionImpl`.

Listing 8-20 com.bea.adapter.dbms.spi.LocalTransactionImpl

```
javax.resource.spi.LocalTransaction
  |
  |-->com.bea.adapter.spi.AbstractLocalTransaction
        |
        |-->com.bea.adapter.dbms.spi.LocalTransactionImpl
```

Developers' Comments

This class utilizes the ADK's abstract superclass which provides logging and event notification. The superclass implements both the CCI and SPI LocalTransaction interfaces provided by Sun. The DBMS adapter's concrete class implements the three abstract methods of the superclass:

- `doBeginTx()`
- `doCommitTx()`
- `doRollbackTx()`

Step 3: Implementing the Common Client Interface Package

To implement the DBMS adapter Common Client Interface (CCI) and meet the J2EE-compliant CCI requirements, ADK classes to create the following concrete classes were extended:

Table 8-7 CCI Class Extensions

This concrete class...	Extends this ADK class...
`ConnectionImpl`	`AbstractConnection`
`InteractionImpl`	`AbstractInteraction`
`InteractionSpecImpl`	`InteractionSpecImpl`

These classes provide connectivity to and access back-end systems. The client interface specifies the format of the request and response records for a given interaction with the EIS.

Note: Although implementing the Common Client Interface (CCI) is optional in the Java Connector Architecture 1.0 specification, it is likely to be required in the future. For your reference, the DBMS adapter provides a complete implementation.

ConnectionImpl

A Connection represents an application-level handle that is used by a client to access the underlying physical connection. The actual physical connection associated with a Connection instance is represented by a ManagedConnection instance. For more information, read about how the ADK implements the AbstractConnection instance in "Connection."

Basic Implementation

The ADK provides `com.bea.adapter.cci.AbstractConnection`, an implementation of the J2EE interface `javax.resource.cci.Connection`. The DBMS adapter in by implementing `com.bea.adapter.dbms.cci.ConnectionImpl`. Listing 8-21 shows the derivation tree for `ConnectionImpl`.

Listing 8-21 com.bea.adapter.dbms.cci.ConnectionImpl

```
javax.resource.cci.Connection
  |
  |-->com.bea.adapter.cci.AbstractConnection
        |
        |-->com.bea.adapter.dbms.cci.ConnectionImpl
```

Developers' Comments

The ConnectionImpl class is the DBMS adapter's concrete implementation of the `javax.resource.cci.Connection interface`. It extends the ADK's AbstractConnection class. The actual physical connection associated with a connection instance is represented by a ManagedConnection instance.

A client gets a connection instance by using the `getConnection()` method on a ConnectionFactory instance. A connection can be associated with zero or more interaction instances. The simplicity of this concrete class is a testament to the power of extending the ADK's base classes.

InteractionImpl

The Interaction enables a component to execute EIS functions. An interaction instance is created from a connection and is required to maintain its association with the Connection instance. For more information, read about how the ADK implements the AbstractInteraction instance in "Interaction."

Basic Implementation

The ADK provides `com.bea.adapter.cci.AbstractInteraction`, an implementation of the J2EE interface `javax.resource.cci.Interaction`. The DBMS adapter extends this class in `com.bea.adapter.dbms.cci.InteractionImpl`. Listing 8-22 shows the derivation tree for `InteractionImpl`.

Listing 8-22 com.bea.adapter.dbms.cci.InteractionImpl

```
javax.resource.cci.Interaction
  |
  |-->com.bea.adapter.cci.AbstractInteraction
        |
        |-->com.bea.adapter.dbms.cci.InteractionImpl
```

Developers' Comments

This is the concrete implementation of the ADK's Interaction object. As expected, the methods are EIS-specific implementations of Java Connector Architecture/ADK required methods.

Both versions of the Java Connector Architecture's `javax.resource.cci.Interactionexecute()`method (the central method of this class) were implemented. The main logic for the `execute()` method has the following

signature: `public Record execute(InteractionSpec ispec, Record input)`. This method return the actual output record from the interaction, so it is the one that is called more often.

The second implementation is provided as a convenience method. This form of `execute()` has the following signature: `public boolean execute(InteractionSpec ispec, Record input, Record output)`. The second implementation's logic returns a boolean, which indicates only the success or failure of the interaction.

InteractionSpecImpl

An InteractionSpecImpl holds properties for driving an interaction with an EIS instance. An InteractionSpec is used by an interaction to execute the specified function on an underlying EIS.

The CCI specification defines a set of standard properties for an InteractionSpec, but an InteractionSpec implementation is not required to support a standard property if that property does not apply to its underlying EIS.

The InteractionSpec implementation class must provide getter and setter methods for each of its supported properties. The getter and setter methods convention should be based on the Java Beans design pattern. For more information, read about how the ADK implements the InteractionSpecImpl instance in "InteractionSpec."

Basic Implementation

The ADK provides `com.bea.adapter.cci.InteractionSpecImpl`, an implementation of the J2EE interface `javax.resource.cci.InteractionSpec`. The DBMS adapter extends this class in `com.bea.adapter.dbms.cci.InteractionSpecImpl`. Listing 8-23 shows the derivation tree for `InteractionSpecImpl`.

Listing 8-23 com.bea.adapter.dbms.cci.InteractionSpecImpl

```
javax.resource.cci.InteractionSpec
  |
  |-->com.bea.adapter.cci.InteractionSpecImpl
       |
       |-->com.bea.adapter.dbms.cci.InteractionSpecImpl
```

Developers' Comments

An implementation class for InteractionSpec interface is required to implement the java.io.Serializable interface. InteractionSpec extends the ADK InteractionSpec in order to add setter/getter methods for the String attribute `m_sql`. The getter/setter methods should be based on the Java Beans design pattern as specified in the Java Connector Architecture 1.0 specification.

Step 4: Implementing the Event Package

This package contains only one class: the DBMS adapter EventGeneratorWorker. It does the work for the event generator for the DBMS adapter.

DbmsEventGeneratorWorker

The EventGeneratorWorker class implements the following interfaces:

- com.bea.wlai.event.IEventGenerator
- java.lang.Runnable.

Basic Implementation

The DBMS adapter EventGeneratorWorker extends the ADK's AbstractPullEventGenerator because a database cannot "push" information to the event generator; you therefore need to "pull" or actually "poll" the database for changes about which you are interested in being notified. Listing 8-24 shows the derivation tree for `DbmsEventGeneratorWorker`.

Listing 8-24 DbmsEventGeneratorWorker

```
com.bea.adapter.event.AbstractEventGenerator
  |
  |-->com.bea.adapter.event.AbstractPullEventGenerator
        |
        |-->com.bea.adapter.dbms.event.DbmsEventGeneratorWorker
```

Developers' Comments

This concrete implementation of the ADK's AbstractPullEventGenerator implements the abstract methods:

- `protected abstract void postEvents(IEventRouter router) throws Exception`
- `protected abstract void setupNewTypes(List listOfNewTypes)`
- `protected abstract void removeDeadTypes(List listOfDeadTypes).`

It also overrides the following methods:

- `void doInit(Map map)`
- `void doCleanUpOnQuit().`

These methods are EIS specific and are used to identify an event within the context of the EIS while interacting with the database to create and remove event definitions and events. Additionally, these methods create and remove the actual triggers on the database that are fired when an event occurs.

The key method of the class is `postEvents()`. It creates the IEvent objects of the data taken from rows in the EVENT table of the database. This method takes an IEventRouter as an argument. After creating an IEvent using the IEventDefinition object's `createDefaultEvent()` method, it populates the event data, and the event is propagated to the router by calling `router.postEvent(event)`. Once the event is sent to the router, the method deletes the row of event data from the database.

The method `setupNewTypes()`creates new event definitions, making sure that the appropriate trigger is created for the database. For each event definition, the method creates a trigger information object that describes the catalog, schema, table, triggerType, and trigger key that the event definition represents. A map of trigger keys is kept so that triggers are not redundantly added to the database. If the map doesn't contain the new key, the trigger text for the database is generated.

The method `removeDeadTypes()` also creates a trigger information object; however, it also checks if one or more event types match it. All event definitions that match this trigger are removed from the map, and then the trigger is removed from the database.

Step 5: Deploying the Adapter

After implementing the SPI, CCI and event interfaces, the adapter is deployed. To deploy the adapter:

- Update the `RA.XML` File.
- Create the `.rar` File.
- Deploy the `.rar` File.

Step 5a: Update the RA.XML File

The DBMS adapter provides the `ra.xml` file in the adapter's `.rar` file (`META-INF/ra.xml`). Since the DBMS adapter extends the AbstactManagedConnectionFactory class, the following properties are provided in the `ra.xml` file:

- LogLevel
- LanguageCode
- CountryCode
- MessageBundleBase
- LogConfigFile
- RootLogContext
- AdditionalLogContext

However, the DBMS sample adapter requires these additional declarations:

Table 8-8 RA.XML Properties

Property	Description
UserName	The username for DBMS adapter login.
Password	The password for DBMS adapter login.
DataSourceName	The name of the JDBC connection pool

Table 8-8 RA.XML Properties

Property	Description
XaCompliantAsBoolean	Indicates whether or not the data source is XA compliant.
PingTable	Identifies the table in the DBMS that can be "pinged" to verify the connection.

You can view the complete `ra.xml` file for the DBMS adapter in:

- `*WLAI_HOME\dev\dbms\src\rar\meta-inf\`

*`WLAI_HOME` is the drive or home directory where BEA WebLogic Application Integration is installed.

Step 5b: Create the .rar File

Class files, logging configuration, and message bundle(s) should be bundled into a file. This `.jar` file and `META-INF/ra.xml` should then be bundled into `.rar` file. The ANT build.xml file demonstrates how to properly construct the `.rar` file.

Step 5c: Deploy the .rar File

The `.rar` file should be deployed into the container in the J2EE-compliant application server. The deployment procedure is different on every J2EE-compliant server.

How the DBMS Adapter Design-Time GUI was Developed

The design-time GUI is the user interface that allows the user to create application views, add services and events and deploy the adapter if it is hosted in the BEA WebLogic Application Integration. This section discusses some specific design time issues that were considered during the development of the DBMS adapter.

The process of creating the DBMS adapter design-time GUI is comprised of the following steps:

- Step 1: Development Considerations
- Step 2: Determine Necessary Java Server Pages
- Step 3: Create the Message Bundle
- Step 4: Implementing the Design-Time GUI
- Step 5: Writing Java Server Pages

Step 1: Development Considerations

Some of the important development considerations regarding the design-time GUI for the DBMS adapter included:

- Determine the database(s) that were going to be supported.
- Determine browsing depth.
- Determine the DBMS schema generation.
- Determine if the adapter should support testing of services and events.

Step 2: Determine Necessary Java Server Pages

The DBMS adapter uses the ADK's Java Server Pages (JSPs) for a design-time GUI; however, additional Java Server pages have been added to provide adapter-specific functionality. A description of the additional JSPs follows:

Filename	Description
`addevent.jsp`	The Add Event page allows a user to add a new event to the application view.
`addservice.jsp`	The Add Service page allows the user to add a new service to the application view.

Filename	Description
`browse.jsp`	The Browse page handles the flow logic and display for the Browse window of the DBMS adapter. Although this functionality was developed specifically for this adapter, it illustrates a fairly common interaction between the design-time interface and the underlying adapter. It uses the DesignTimeRequestHandler (handler) of the DBMS adapter, which extends the ADK's AbstractDesignTimeRequestHandler. The best way to understand the browse functionality of the DBMS adapter is to deploy the adapter and use your web browser to access the design-time framework.
`confconn.jsp`	The Confirm Connection page provides a form for a user to specify connection parameters for the EIS.
`display.jsp`	The Display page is the main "displayer" for other Java Server Pages in the adapter. It authenticates the request and sets the basic look of each screen with a title and description. It includes each JSP that is requested and displays it inline.
`testform.jsp`	The Testform page is included (<jsp:include page='testform.jsp'/>) in the ADK's testsrvc.jsp page. It accesses the InteractionSpec for this interaction and displays the SQL for the service on screen. It then creates a form for gathering required user input to test a service. It does this by getting the RequestDocumentDefinition from the handler's ApplicationView and then passing it along with the `.jsp` Writer to a utility class, com.bea.adapter.dbms.utils.TestFormBuilder, which actually creates the required form.

Step 3: Create the Message Bundle

To support the internationalization of all text labels, messages and exceptions, the DBMS adapter uses a message bundle based on a text property file. The property file uses copied name value pairs from the `BEA_WLS_SAMPLE_ADK` property file, and new entries were added for properties specific to the DBMS adapter.

The message bundle for the DBMS adapter is contained in *`wlaihome\dbms\src` directory, which was installed with the ADK. The "`wlaihome`" indicates the drive used during installation. Please refer to `BEA_WLS_DBMS_ADK.properties` in the directory above.

For additional instructions on creating a message bundle, refer to the JavaSoft tutorial on internationalization at http://java.sun.com/docs/books/tutorial/i18n/index.html.

Step 4: Implementing the Design-Time GUI

To implement the design-time GUI, you need to create a DesignTimeRequestHandler class. This class accepts user input from a form and provides methods to perform a design-time action. For more information on the DesignTimeRequestHandler, see "Step 4: Implementing the Design-Time GUI" in "Developing a Design-Time GUI."

The DBMS adapter public class DesignTimeRequestHandler extends AbstractDesignTimeRequestHandler, and it provides the methods shown in Table 8-9:

Table 8-9 Methods for the DBMS Sample Adapter Design-Time GUI

Method	Description
`browse(java.lang.String dbtype, com.bea.connector.DocumentRecord input)`	Handles the back-end behavior for the "Browse" functionality of the `addevent.jsp` and `addservc.jsp`.
`getAdapterLogicalName()`	Returns the adapter's logical name and helps parent when deploying application views, etc.
`getManagedConnectionFactoryClass()`	Returns the adapter's SPI ManagedConnectionFactory implementation class, used by parent to get a CCI connection to the EIS.
`supportsServiceTest()`	Indicates that this adapter design time supports the testing of services.

Method	Description
`initServiceDescriptor(ActionResult result, IServiceDescriptor sd, HttpServletRequest request)`	Initializes a service descriptor which involves creating the request and response schemas for the service. A typical approach is to execute an Interaction against the EIS to retrieve metadata and transform it into an XML schema. Consequently, the CCI interface provided by the adapter was used. This method is called from the "addsrvc" method of the AbstractDesignTimeRequestHandler.
`initEventDescriptor(ActionResult result, IEventDescriptor ed, HttpServletRequest request)`	Initializes an event descriptor. The event descriptor provides information about an event on an application view. Subclasses will need to supply an implementation of this method. If events are not supported, then the implementation should throw an UnsupportedOperationException. This method will not be called (by the AbstractDesignTimeRequestHandler) until the event name and definition have been validated and it is confirmed that the event does not already exist for the application view.
`GetDatabaseType()`	This method is used to determine the type of dbms being used. At present BEA WebLogic Application Integration supports Oracle, Microsoft SQL Server, and Cloudscape.

Step 5: Writing Java Server Pages

The following issues are relevant for the DBMS adapter, and you may encounter them as you develop your own adapter.

- Custom JSP Tags
- Save an Object's State
- Write the WEB-INF/web.xml Web Application Deployment Descriptor

Custom JSP Tags

The Java Server Pages are displayed within the `display.jsp`; thus `display.jsp` is the first `.jsp` that needs to be written. The ADK provides a library of custom `.jsp` tags, which are used extensively throughout the Java server pages of the ADK and DBMS adapter. They provide the ability to add validation, to save field values when the user clicks away, and a number of other features.

Save an Object's State

You may also need to save an object's state as you write the Java server pages. There are a number of ways to save an object's state when building your adapter using the ADK. The AbstractDesignTimeRequestHandler maintains an ApplicationViewDescriptor of the application view being edited. This is often the best place to save state. Calls to the handler are fast and efficient.

You can also ask the AbstractDesignTimeRequestHandler for a Manager Bean, using its convenience methods: getApplicationViewManager(); getSchemaManager(); and getNamespaceManager(), to retrieve information from the repository about an application view. This is more time-consuming but may be necessary on occasion. Since it is a Java Server Page, you can also use the session object, although everything put in the session must explicitly implement the java.io.serializable interface.

Write the WEB-INF/web.xml Web Application Deployment Descriptor

Write the `WEB-INF/web.xml` Web application deployment descriptor. In most cases, you should use the sample adapter's web.xml file as a starting point and modify the necessary components to fit your needs. For detailed information, see the BEA WebLogic Server product documentation at HTTP://edocs.bea.com.

APPENDIX

C The eMail Sample Adapter

This section contains information on the following subjects:

- Introduction to the eMail Sample Adapter
- How the eMail Sample Adapter Works
- How the eMail Event Adapter was Developed
- Creating the eMail Adapter Design-Time GUI

Introduction to the eMail Sample Adapter

The eMail adapter is a J2EE-compliant adapter built with BEA WebLogic Application Integration Adapter Development Kit. The purpose of the eMail adapter is to provide a way for any application to send notice in case of system failure or process completion. This notification is directed using email, which could be configured to target multiple addresses or even a pager. A single templated message could be created for numerous errors allowing the adapter to plug in replaceable parameters and send the notification.

The eMail adapter provides sample implementations of both a services and events. The event implementation provides sample code for both push and pull event generator paradigms. The service implementation enables the client to send and eMail message

with a minimum of information. Service-specific data is information required to send an email message, such as source address, target addresses, subject, and the body of a message.

The eMail sample adapter gives you (developers and business analysts) a concrete example of an adapter, including a JSP-based GUI, to help you understand the possibilities that are at your disposal using the ADK to build adapters. If you are a business analyst, you may enjoy running through the interface to get a better understanding of an "application view", "service", and "event" as shown in "How the eMail Sample Adapter Works."

If you are an adapter developer, you will also want to review "How the eMail Event Adapter was Developed" and "Creating the eMail Adapter Design-Time GUI", the code, and javadoc to gain insight into how you can extend and use the classes of the ADK to build a JCA-compliant adapter.

How the eMail Sample Adapter Works

This section provides you with an opportunity to have a look at the eMail adapter before you start developing an adapter of your own. If you are a business analyst, you may enjoy running through the interface to get a feel for how the adapter works. The example in this section shows how to create an application view for sending or receiving emails. This section contains information on the following subjects:

- Before You Begin
- Accessing the eMail Sample Adapter
- A Tour of the eMail Sample Adapter

Before You Begin

Make sure the following tasks have been performed before you try to access the eMail sample adapter:

- Install the BEA WebLogic Application Integration. For more information, see the *BEA WebLogic Application Integration Installation and Setup Guide.*

- Set up the ADK Ant-Based Make Process. For more information, see "Step 2c: Setting Up the Build Process."
- Ensure that the eMail sample adapter has been deployed so that the design-time GUI is accessible. For more information, see the *BEA WebLogic Application Integration Installation and Setup Guide*.

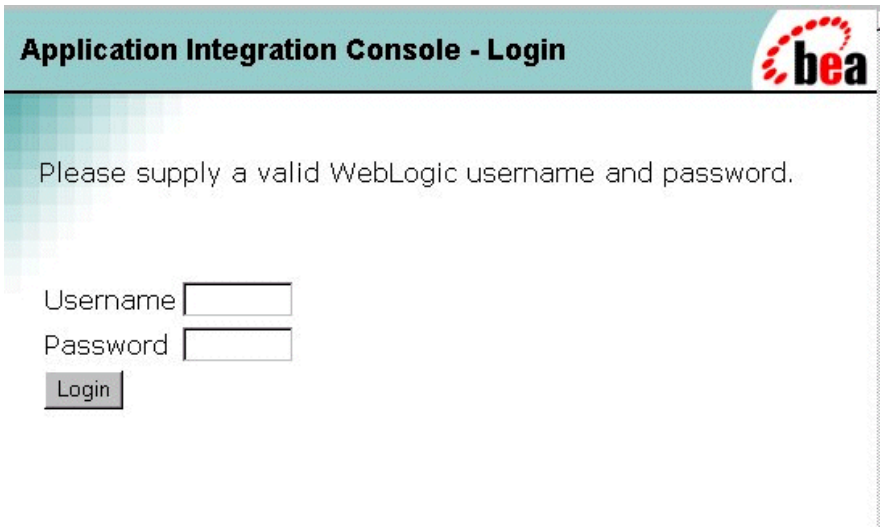
Accessing the eMail Sample Adapter

To access the eMail Adapter:

1. Open a new browser window.
2. Open the URL for your system's Application View Management Console.

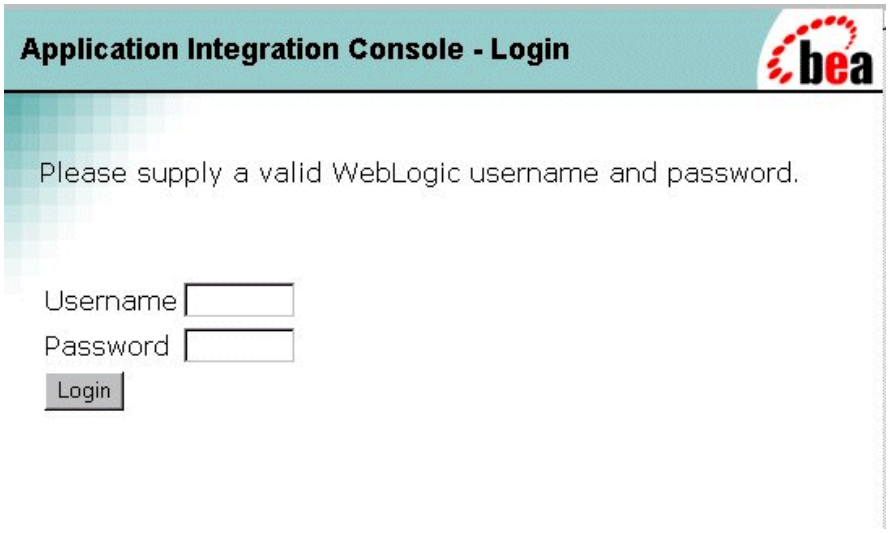
   `http://<HOSTNAME>:7601/wlai`

   The Application Integration Console - Login page displays.

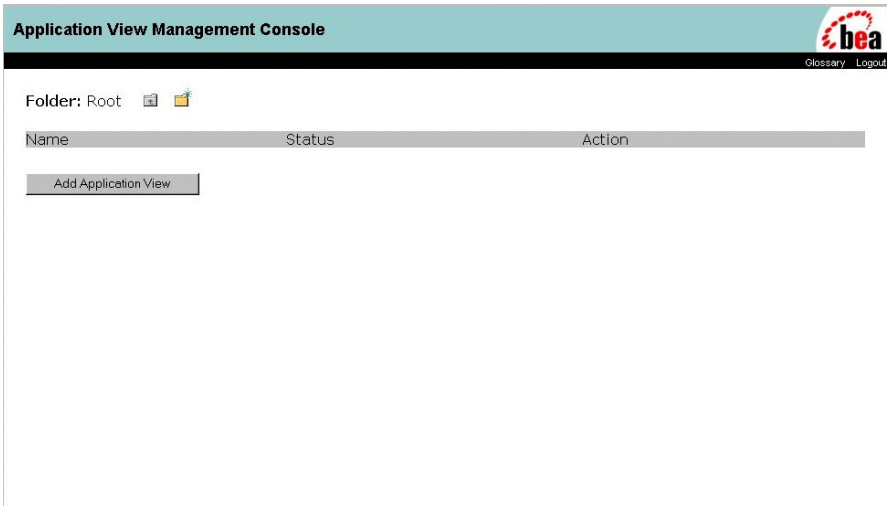


A Tour of the eMail Sample Adapter

This section provides you with a short tour through the eMail Sample adapter. Before you begin, you need to have the eMail adapter Login page up on your browser. For information about accessing the eMail adapter, see "Accessing the eMail Sample Adapter."

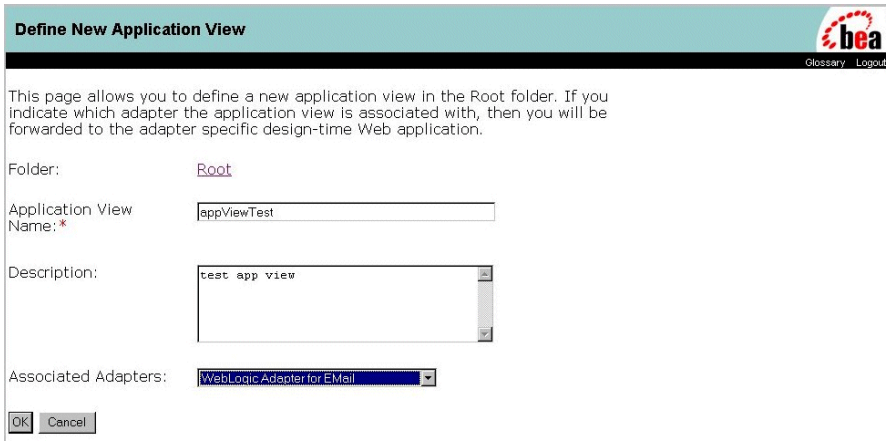


1. To log on to the Application View Management Console, enter your WebLogic username and password, then click Login. The Application View Management Console displays.



2. Click Add Application View. The Define New Application View page displays. When you create the application view, you provide a description that associates the application view with the eMail adapter.

Note: For detailed information about application views, see "Introduction to Using Application Integration." For detailed information about defining application views, see "Defining Application Views" in the *BEA WebLogic Application Integration User Guide*.



3. To define an application view:

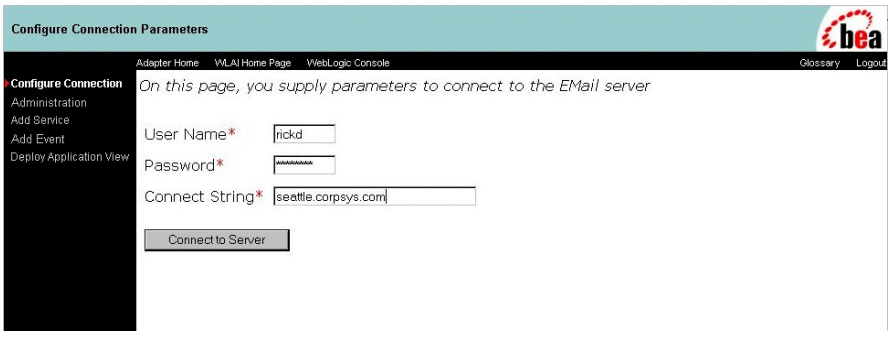
   a. In the Application View Name field, enter TestAppView.

Note: The name should describe the set of functions performed by this application. Each application view name must be unique to its adapter. Valid characters are anything except '.', '#', '\', '+', '&', ',', ''', '"", and a space.

   b. In the Description field, enter a brief description of the application view.

   c. From the Associated Adapters list, choose the eMail adapter to use to create this application view.

   d. Click OK. The Configure Connection Parameters page displays.



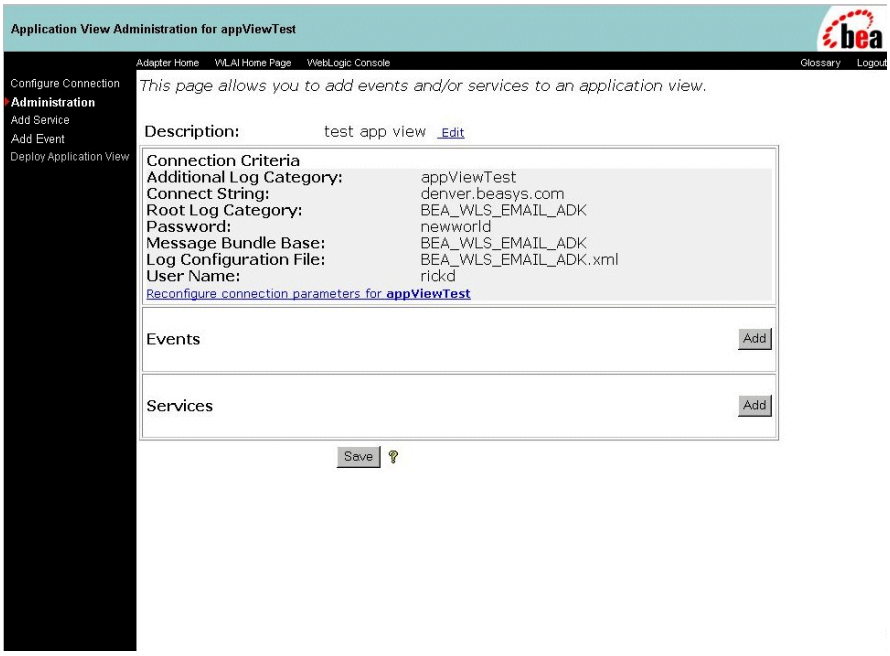
4. At the Configure Connection Parameters page, you define the network-related information necessary for the application view to interact with the target EIS. You need to enter this information only once per application view:

   a. Enter your eMail User Name and eMail Password.

   b. Enter the eMail service URL in the Connect String field.

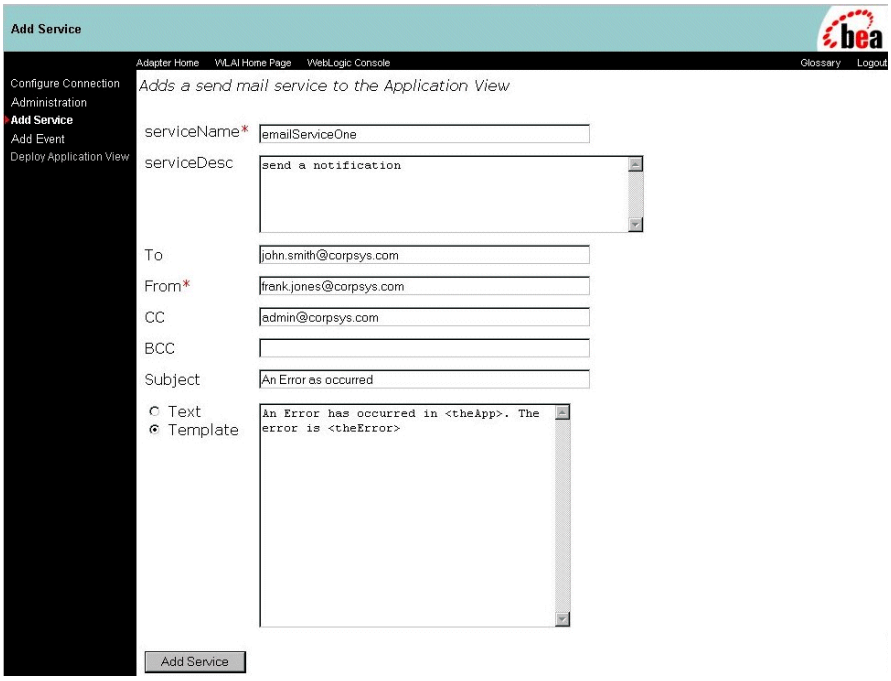
   c. Click Continue. The Application View Administration page displays.

   Note: The Administration page summarizes the connection criteria, and once events and services are defined, you can view the schemas and summaries and also delete an event or service from this page.



5. Now that you have created an application view, you are ready to add a service. To add a service:

   a. In the Administration page, click Add in the Services field. The Add Service page displays.



b. In the serviceName field, a meaningful name for the service.

c. In the serviceDesc field, enter a user description for the service.

d. In the To field, enter a list of target email addresses.

e. In the From field, enter the source email address.

f. In the CC filed, enter a list of email addresses to receive a copy.

g. In the BCC field, enter a list of email addresses to receive a blind copy, delimited by a semicolon.

h. In the Subject Field, enter the subject of the email.

i. Select the Text radio button to send a plain text message. Select Template to define replaceable parameters.

Note: The body type can be either text or template. A template can contain tags for replaceable parameters.

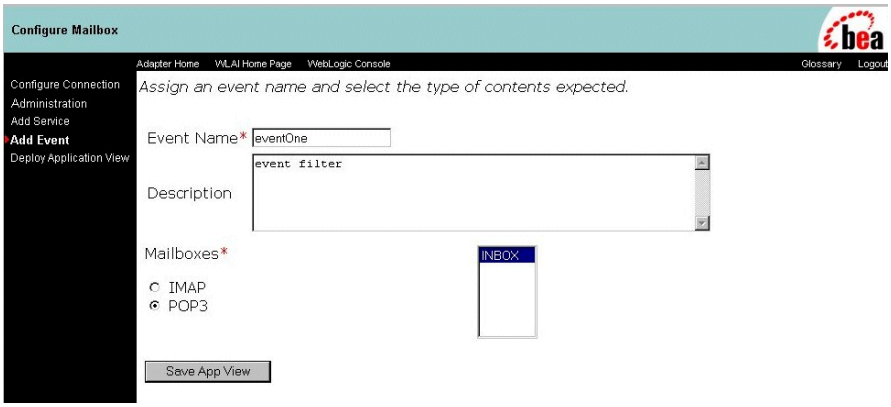
j. In the open text field, enter the text of the message.

Note: The email body can contain replaceable parameters if the type is template, otherwise it will contain a text message.

k. Click Add Service. The Administration page displayed.

6. Now that you have created an application view, you are ready to add an event to it. To add an event:

   a. In the Administration page, click Add in the Event field. The Add Event page displays.



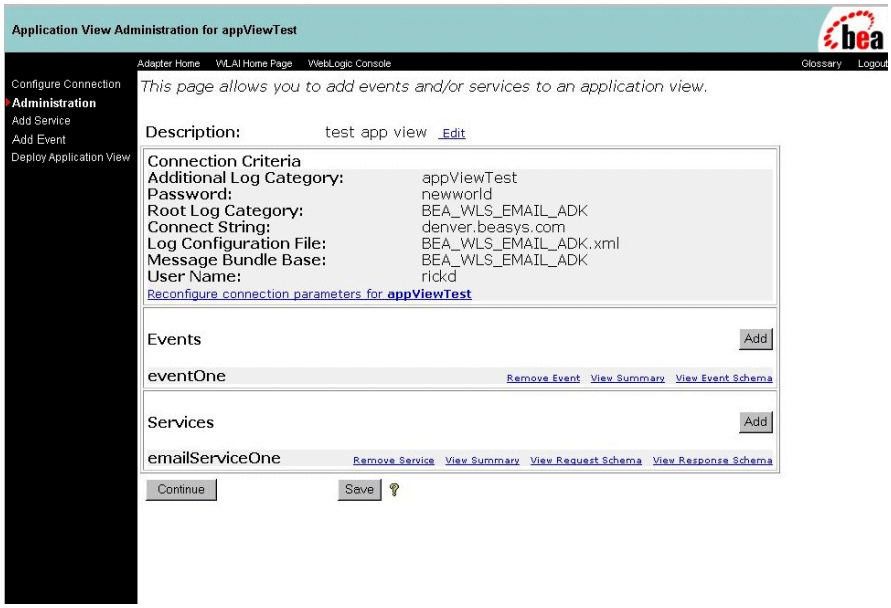
   b. In the eventName field, enter a meaningful name for the event.

   c. In the eventDesc field, enter a description of the event.

   d. Select either the IMAP or POP3 radio button. When configuring an event you can use either the POP3, or IMAP access protocols depending on the type of event generator you wish to deploy. Use IMAP if you are trying to deploy the Push event generator. Use POP3 to deploy the Pull event generator. When IMAP is selected you can select a folder to listen to. POP3/Pull supports the use of a single folder, the INBOX folder.

   e. Scroll to select a folder to query for mail.

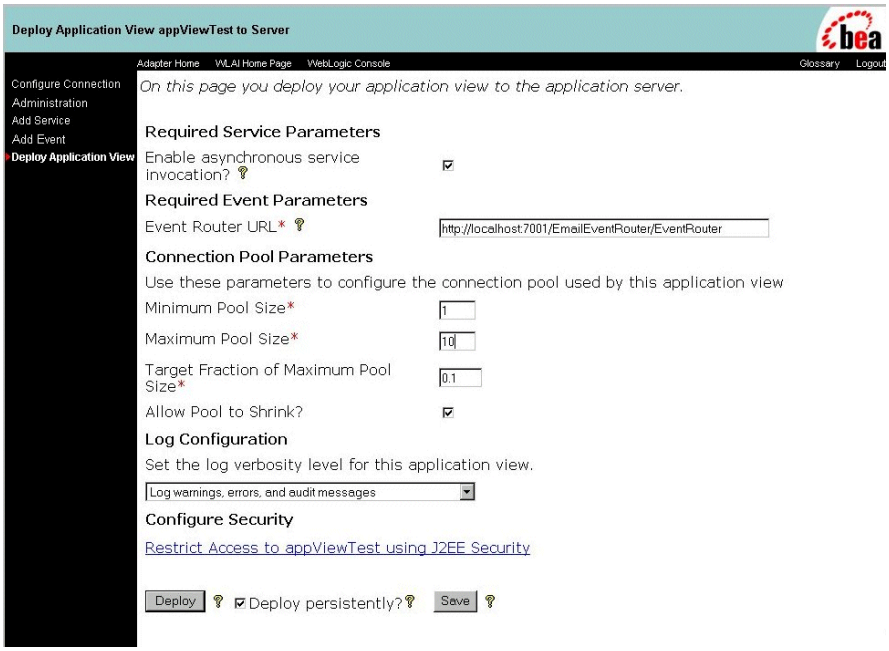
   f. Click Save AppView. The Appliation View Administration page displays.



7. Prepare to deploy the application view. The Application View Administration page provides you with a single location for confirming the content of your application view before you save it or deploy it. In this page, you can view the following:

   - Confirm or edit the description of the application view.
   - Confirm or reconfigure Connection Criteria for the application view.
   - Delete services and events.
   - Save the application view so you can return to it later or deploy the application view to the server.

   After verifying the application view parameters, click Continue. The Deploy Application View to Server page displays.



8. Deploy the Application View. In order to deploy the application view, you must provide several parameters such as enabling asynchronous service invocation, providing the event router URL, and changing the connection pool parameters, among other parameters.

   To deploy the application view:

   a. Make sure the Enable Asynchronous Service Invocation check box is checked.

   b. In the Event Router URL field, enter `http://<HOSTNAME>:<PORT>/emailEventRouter/EventRouter`

   c. For the Connection Pool Parameters, accept the default values:

      Minimum Pool Size - 1

      Maximum Pool Size - 10

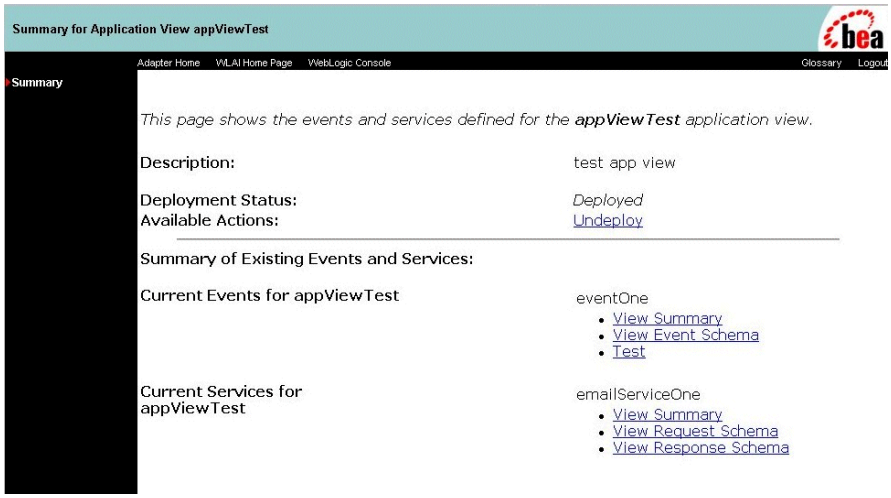
      Target Fraction of Maximum Pool Size - 0.7

      Allow Pool to Shrink - checked

d. In the Log Configuration field, enter Log warnings, errors, and audit messages.

e. Make sure the Deploy persistently? box is checked.

f. Click Deploy.



9. Once the application view is deployed, the summary page displays all relevant information about the deployed application view. Use the Summary page to view schemas, event and service summaries, test services and events and undeploy the application view.

How the eMail Event Adapter was Developed

This section describes each interface used to develop the eMail adapter. The ADK provides many of the necessary implementations required by a Java Connector Architecture-compliant adapter; however, since some interfaces cannot be fully implemented until the EIS and its environment are defined, the eMail adapter was created to illustrate the detail-specific or concrete implementation of the abstract classes provided in the ADK.

The process of creating the eMail adapter is comprised of the following steps:

- Development Reference Documentation
- Step 1: Development Considerations
- Step 2: Implementing the Server Provider Interface Package
- Step 3: Implementing the Common Client Interface Package
- Step 4: Implementing the Event Package
- Step 5: Deploying the Adapter

Development Reference Documentation

You can review the Javadoc and code for the methods defined in the steps that follow in this section to see how the implementations provided by the ADK were leveraged. You can find the Javadoc for this implementation in:

`<WLAI_HOME>\dev\email\docs\api\index.html`

You can find the code listing for this package in:

`<WLAI_HOME>\dev\email\src\email\spi`

Note: `<WLAI_HOME>` is the drive or home directory where BEA WebLogic Application Integration is installed.

Step 1: Development Considerations

The "Adapter Setup Worksheet" is available to help adapter developers identify and collect critical information about an adapter they are developing before they begin coding. For the email sample adapter, the worksheet questions are answered as follows:

Note: Questions preceded by an asterisk (*) are required to use the GenerateAdapterTemplate utility.

1. **What is the name of the EIS for which you are developing an adapter?*

   email API

2. **What is the version of the EIS?*

   n/a

3. **What is the type of EIS; for example, DBMS, ERP, etc.?*

   email API

4. **Who is the vendor name of this adapter?*

   BEA

5. **What is the version number for this adapter?*

   None - Sample Only

6. **What is the adapter logical name?*

   BEA_WLS_EMAIL

7. *Does the adapter need to invoke functionality within the EIS?*

   Yes

   If so, then your adapter needs to support services.

   Yes

8. *What mechanism/API is provided by the EIS to allow an external program to invoke functionality provided by the EIS?*

   It is an API.

9. *What information is needed to create a session/connection to the EIS for this mechanism*?

   Need to acquire a session, and from the session you can get a transport object. The transport will be used to send mail.

10. *What information is needed to determine which function(s) will be invoked in the EIS for a given service?*

    Javadoc for eMail API.

11. *Does the EIS allow you to query it for input and output requirements for a given function?*

    No

    If so, what information is needed to determine the input requirements for the service?

    n/a

12. *For all the input requirements, which ones are static across all requests? Your adapter should encode static information into an InteractionSpec object.*

    To, From, CC, BCC, Subject, Body, Type

13. *For all the input requirements, which ones are dynamic per request? Your adapter should provide an XML schema that describes the input parameters required by this service per request.*

    To, From, CC, BCC, Subject, Body, Type

14. *What information is needed to determine the output requirements for the service?*

    Success or failure of a send call if in error. Need to extract the error and any email addresses in the error.

15. *Does the EIS provide a mechanism to browse a catalog of functions your adapter can invoke? If so, your adapter should support browsing of services.*

    No

16. *Does the adapter need to receive notifications of changes that occur inside the EIS? If so, then your adapter needs to support events.*

    Yes. Need to provide examples of both types of events.

17. *What mechanism/API is provided by the EIS to allow an external program to receive notification of events in the EIS? The answer of this question will help determine if a pull or a push mechanism is developed.*

    Can either poll a folder for new messages or add a listener (IMAP) to a folder for new messages.

18. *Does the EIS provide a way to determine which events your adapter can support?*

    No

19. *Does the EIS provide a way to query for metadata for a given event?*

    Some

20. *What locales (language/country) does your adapter need to support?*

    English

Step 2: Implementing the Server Provider Interface Package

To implement the eMail adapter Server Provider Interface (SPI) and meet the J2EE-compliant SPI requirements, the classes in the ADK were extended to create the following concrete classes:

Table 8-10 SPI Class Extensions

This concrete class...	Extends this ADK class...
`ManagedConnectionFactoryImpl`	`AbstractManagedConnectionFactory`
`ManagedConnectionImpl`	`AbstractManagedConnection`
`ConnectionMetaDataImpl`	`AbstractConnectionMetaData`

These classes provide connectivity to an EIS could be used to establish transaction demarcation, and allow management of a selected EIS.

ManagedConnectionFactoryImpl

The first step in implementing an SPI for the email adapter was to implement the ManagedConnectionFactory interface. A ManagedConnectionFactory supports connection pooling by providing methods for matching and creating a ManagedConnection instance.

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractManagedConnection Factory`, an implementation of the Java Connector Architecture interface `javax.resource.spi. ManagedConnectionFactory`. The eMail adapter extends this class in `email.spi.ManagedConnectionFactoryImpl`. Listing 8-25shows the derivation tree for `ManagedConnectionFactoryImpl`.

Listing 8-25 com.bea.adapter.email.spi.ManagedConnectionFactory Impl

```
javax.resource.spi.ManagedConnectionFactory
  |
  |-->com.bea.adapter.spi.AbstractManagedConnectionFactory
        |
        |-->email.spi.ManagedConnectionFactoryImpl
```

Developers' Comments

ManagedConnectionFactory creates physical connections to an underlying EIS for the application server. A physical connection is encapsulated by a ManagedConnection instance.

ManagedConnectionFactoryImpl is a factory for both ManagedConnections and adapter specific connectionFactory instances. The eMail adapter has a simple implementation for this factory object. Four methods were implemented from the base classes, two of which are abstract. The abstract methods are `createConnectionFactory()` and `createManagedConnection()`. Both of these implementations return adapter-specific object instances. The concrete methods overridden by the eMail adapter include `checkState()` and `hashCode()`. The implementation of `checkState()` validates the connection parameters required for the adapter to acquire a physical connection. The implementation of `hashCode()` is also based on connection parameters specific to the eMail adapter.

ManagedConnection

A ManagedConnection instance represents a physical connection to the underlying EIS in a managed environment. ManagedConnection objects are pooled by the application server. For more information, read about how the ADK implements the AbstractManagedConnection instance in "ManagedConnection."

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractManagedConnection`, an implementation of the J2EE interface `javax.resource.spi.ManagedConnection`. The eMail adapter extends this class in `email.spi.ManagedConnectionImpl`. Listing 8-26 shows the derivation tree for `ManagedConnectionImpl`.

Listing 8-26 com.bea.adapter.email.spi.ManagedConnectionImpl

```
javax.resource.spi.ManagedConnection
  |
  |-->com.bea.adapter.spi.AbstractManagedConnection
       |
       |-->email.spi.ManagedConnectionImpl
```

Developers' Comments

The ManagedConnectionImpl represents the physical connection to the EIS. The eMail adapter overrides what is probably the minimum required functionality of the base classes. There are two abstract methods and two concrete methods that the eMail adapter implements: `getConnection()` and `createMetaData()`.

The method `getConnection()` is used to wrap the current ManagedConnection with a ConnectionImpl and return it to the caller. The value for myCredentials is compared with the connectionRequestInfo passed. If they match, the current ManagedConnection is wrapped with a ConnectionImpl. The `createMetaData()` method simply instantiates and returns a ConnectionMetaDataImpl.

The other two methods, `destroyPhysicalConnection()` and `compareCredentials()`, are overridden because they are either too simple or empty in the base class. These are both concrete methods in the base class. The method `destroyPhysicalConnection()` is adapter specific; this method is used to free resources associated with acquiring a physical connection.

The `compareCredentials()` method is used by `matchManagedConnections()` method in the ManagedConnectionFactory. The `matchManagedConnections()` method tries to associate a request for connection with an existing connection matching the same criteria. The criteria is defined in the `compareCredentials()` method. Usernames are used by the eMail adapter as the criteria.

ConnectionMetaDataImpl

The ManagedConnectionMetaData interface provides information about the underlying EIS instance associated with a ManagedConnection instance. An application server uses this information to get runtime information about a connected EIS instance. For more information, read about how the ADK implements the AbstractConnectionMetaData instance in "ConnectionMetaData."

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractConnectionMetaData`, an implementation of the J2EE interface `javax.resource.spi.ManagedConnection MetaData`. The eMail adapter extends this class in `email.spi.ConnectionMetaDataImpl`. Listing 8-27 shows the derivation tree for `ConnectionMetaDataImpl`.

Listing 8-27 com.bea.adapter.email.spi.ConnectionMetaDataImpl

```
javax.resource.spi.ManagedConnectionMetaData
  |
  |-->com.bea.adapter.spi.AbstractConnectionMetaData
       |
       |-->email.spi.ConnectionMetaDataImpl
```

Developers' Comments

The ConnectionMetaDataImpl class provides metadata for an EIS. The metadata implementation describes very specific data required by the application server. The eMail adapter provides an implementation for the abstract methods declared in the base class. These methods provide product name, product version, user name, and max connections allowed.

Step 3: Implementing the Common Client Interface Package

To implement the eMail adapter Common Client Interface (CCI) and meet the J2EE-compliant CCI requirements, classes in the ADK to create the following concrete classes were extended

Table 8-11 CCI Class Extensions

This concrete class...	Extends this ADK class...
`ConnectionImpl`	`AbstractConnection`
`InteractionImpl`	`AbstractInteraction`
`InteractionSpecImpl`	`InteractionSpecImpl`

These classes provide connectivity to and access back-end systems. The client interface specifies the format of the request and response records for a given interaction with the EIS.

Note: Although implementing the Common Client Interface (CCI) is optional in the Java Connector Architecture 1.0 specification, it is likely to be required in the future. To be prepared, the eMail adapter provides a complete implementation.

ConnectionImpl

A Connection represents an application-level handle that is used by a client to access the underlying physical connection. The actual physical connection associated with a Connection instance is associated with a ManagedConnection instance. For more information, read about how the ADK implements the AbstractConnection instance in "Connection."

Basic Implementation

The ADK provides `com.bea.adapter.cci.AbstractConnection`, an implementation of the J2EE interface `javax.resource.cci.Connection`. The eMail adapter extends this class in `email.cci.ConnectionImpl`. Listing 8-28 shows the derivation tree for `ConnectionImpl`.

Listing 8-28 com.bea.adapter.email.cci.ConnectionImpl

```
javax.resource.cci.Connection
  |
  |-->com.bea.adapter.cci.AbstractConnection
        |
        |-->email.cci.ConnectionImpl
```

Developers' Comments

The ConnectionImpl class is an application-level handle used to access EIS-level resources and functionality. For the eMail adapter the implementation is simple. Derived-functionality was used for all methods except the `createInteraction()` method. This method is an abstract method provided in the connection interface, and unless you have specific needs, this is usually the only method that needs to be defined/overridden. For implementation, you need to return an application-level interaction object.

InteractionImpl

The Interaction enables a component to execute EIS functions. An interaction instance is created from a connection and is required to maintain its association with the Connection instance. For more information, read about how the ADK implements the AbstractInteraction instance in "Interaction."

Basic Implementation

The ADK provides `com.bea.adapter.cci.AbstractInteraction`, an implementation of the J2EE interface `javax.resource.cci.Interaction`. The eMail adapter extends this class in `email.cci.InteractionImpl`. Listing 8-29 shows the derivation tree for `InteractionImpl`.

Listing 8-29 com.bea.adapter.email.cci.InteractionImpl

```
javax.resource.cci.Interaction
  |
  |-->com.bea.adapter.cci.AbstractInteraction
       |
       |-->email.cci.InteractionImpl
```

Developers' Comments

An Interaction enables a component to execute EIS functions. The InteractionImpl class wraps EIS-specific functionality. Using the ConnectionImpl, you can use the physical EIS connection to provide application-level interfaces to the EIS. This is probably where you will spend most of your time.

The two `execute()` methods process according to the method being called and either return an output document in the parameter list or as a result of the call. The last method is `close()`. The `close()` method is used to free resources created in the execution of an EIS call. The `execute()` method creates an email message based on data from both the InteractionSpecImpl and the input DocumentRecord. The data extracted is used to populate a MimeMessage object and is transported according to the internet address data contained. If an error is encountered it is returned in the output DocumentRecord.

InteractionSpecImpl

An InteractionSpecImpl holds properties for driving an interaction with an EIS instance. An InteractionSpec is used by an interaction to execute the specified function on an underlying EIS.

The CCI specification defines a set of standard properties for an InteractionSpec, but an InteractionSpec implementation is not required to support a standard property if that property does not apply to its underlying EIS.

The InteractionSpec implementation class must provide getter and setter methods for each of its supported properties. The getter and setter methods convention should be based on the Java Beans design pattern. For more information, read about how the ADK implements the InteractionSpecImpl instance in "InteractionSpec."

Basic Implementation

The ADK provides `com.bea.adapter.cci.InteractionSpecImpl`, an implementation of the J2EE interface `javax.resource.cci.InteractionSpec`. The eMail adapter extends this class in `email.cci.InteractionSpecImpl`. Listing 8-30 shows the derivation tree for `InteractionSpecImpl`.

Listing 8-30 com.bea.adapter.email.cci.InteractionSpecImpl

```
javax.resource.cci.InteractionSpec
  |
  |-->com.bea.adapter.cci.InteractionSpecImpl
        |
        |-->email.cci.InteractionSpecImpl
```

Developers' Comments

The InteractionSpecImpl class provides properties used in the request to a service. In the case of the eMail adapter the properties are specific to an email message; for example: "To"; "From"; "Subject" etc. The InteractionSpecImpl is very much adapter specific. The data required to fulfill a request varies according to the request, and there are no abstract methods that need to be implemented.

Step 4: Implementing the Event Package

Some utility classes were created to help with implementation. These classes were extended from the ADK classes to the create the following concrete classes:

Table 8-12 Event Class Extensions

This concrete class...	Extends the ADK class...
`EmailEventMetaData`	`EventMetaData`
`EmailPushEvent`	`PushEvent`
`EmailPushHandler`	`java.lang.Object`
`PullEventGenerator`	`AbstractPullEventGenerator`
`PushEventGenerator`	`AbstractPushEventGenerator`

EmailEventMetaData

The ADK provides `com.bea.adapter.event.EventMetaData`, an implementation of the `java.lang.Object`. The eMail adapter extends this class by implementing `email.event.EmailEventMetaData`. Listing 8-31 shows the derivation tree for `EmailEventMetaData`.

Listing 8-31 EmailEventMetaData

```
com.bea.adapter.event.EventMetaData
  |
  |-->email.event.EmailEventMetaData
```

Developers' Comments

The EmailMetaData is used to pass information between the event generator and the handler.

EmailPushEvent

The ADK provides `com.bea.adapter.event.PushEvent`, an implementation of the `java.util.EventObject`. The eMail adapter extends this class by implementing `email.event.EmailPushEvent`. Listing 8-32 shows the derivation tree for `EmailPushEvent`.

Listing 8-32 EmailPushEvent

```
java.util.EventObject
  |
  |-->com.bea.adapter.event.PushEvent
        |
        |-->email.event.EmailPushEvent
```

Developers' Comments

The EmailPushEvent is used to send notification from the handler to the event generator.

EmailPushHandler

The EmailPushHandler extends implements IPushHandler and is the point of contact for the Email EIS. Listing 8-33 shows the derivation tree for `EmailPushHandler`.

Listing 8-33 EmailPushHander

```
com.bea.adapter.event.IPushHandler
  |
  |-->email.event.EmailPushHandler
```

Developers' Comments

The EmailPushHandler implements the ADK interface IPushHandler. The handler interface is provided to abstract EIS event generation from event routing functionality. This is not enforced since the interfaces provided are not required to implement the Push functionality.

The EmailPushHandler implements three interfaces:

- MessageCountListener
- Runnable
- IPushHandler

The only method implemented outside of the scope of the interface methods is `verifyConnection()`. The `verifyConnection()` method validates the connection to the EIS. It does nothing more than check to see if it is connected to the server.

One method of interest is the `run()` method. A thread was implemented in order to poll the folder for message count. Sun's implementation of the IMAP access protocol does not send notification without this polling, so this it does not provide good example of push generation. However, the idea is to show how to separate the generation functionality from the routing functionality. The rest of the implementation is fairly straightforward and follows the interfaces implemented.

PullEventGenerator

The ADK provides `com.bea.adapter.event.AbstractPullEventGenerator`, an implementation of the `java.lang.Object`. The eMail adapter extends this class in `email.event.PullEventGenerator`. Listing 8-34 shows the derivation tree for `PullEventGenerator`.

Listing 8-34 PullEventGenerator

```
com.bea.adapter.event.AbstractEventGenerator
  |
  |-->com.bea.adapter.email.event.AbstractPullEventGenerator
    |
    |-->email.event.PullEventGenerator
```

Developers' Comments

The Email Pull Event Generator is a POP3-only event generator. The reason for this is that POP3 does not allow notifications to be received when a listener is added to the Inbox folder. In order to deploy the PullEventGenerator you need to modify some of the properties contained in the `EmailEventRouter web.xml` file. Once you have the correct properties, the `EmailEventRouter.war` file can be created using the ANT build process.

The Email PullEventGenerator supports a single event type, which is the notification of an email being received in the Inbox folder using the POP3 access protocol. As such, the email event generator probably doesn't need to implement `setupNewTypes()` and `removeDeadTypes()`; however, the event engine will give notification when event types are removed.

Other than the implementation of `setupNewTypes()` and `removeDeadTypes()`, the only other abstract method is `postEvents()`. The `postEvents()` method is the fulcrum to the event generation process. This is where you would add EIS-specific implementations. The email event generator uses the `postEvents()` method to read from the Inbox and route new messages to any listeners.

One other method of interest is the `doCleanUpOnQuit()` method. This method provides a place to free any resources allocated in the event generation process. The email event generator uses `doCleanUpOnQuit()` to free the mail store and release the mail session.

PushEventGenerator

The ADK provides `com.bea.adapter.event.AbstractPullEventGenerator`, an implementation of the `java.lang.Object`. The eMail adapter extends this class in `email.event.PushEventGenerator`. Listing 8-35 shows the derivation tree for `PushEventGenerator`.

Listing 8-35 PushEventGenerator

```
com.bea.adapter.event.AbstractEventGenerator
  |
  |-->com.bea.adapter.email.event.AbstractPushEventGenerator
    |
    |-->email.event.PushEventGenerator
```

Developers' Comments

The Email Push Event Generator is an IMAP only event generator. It is a sample of the push event paradigm. Where the Pull Event Generator uses a thread to continuously poll for an event, the push methodology listens for an event to have been posted. If you look closely at the push event implementation, you will see that it uses a thread to process events in the EmailPushEventHandler. A thread is not necessary to implement the push event. A separate thread was used to implement the push generator.

Additionally, three other classes were used in the push implementation. These are:

- `EmailPushHandler`
- `EmailPushEvent`
- `EmailMetaData`

The EmailPushHandler serves to abstract the push event generation functionality from the event routing. The EmailPushEvent is used to send notification from the handler to the event generator. The EmailMetaData is used to pass information between the event generator and the handler. If you look closely at the PushEventGenerator code, you will find that it knows almost nothing of the EIS. It uses the `setNewTypes()` and `removeDeadTypes()` to create the array it needs to process events, and it uses `postEvents()` to process notifications.

Step 5: Deploying the Adapter

After implementing the SPI, CCI and event interfaces, the adapter was deployed. To deploy the adapter:

- Update the `RA.XML` File
- Create the .rar File
- Deploy the `.rar` File

Step 5a: Update the RA.XML File

The eMail adapter provides the `ra.xml` file in the adapter's `.rar` file (`META-INF/ra.xml`). Since the eMail adapter extends the AbstactManagedConnectionFactory class, the following properties were provided in the `ra.xml` file:

- LogLevel
- LanguageCode
- CountryCode
- MessageBundleBase
- LogConfigFile
- RootLogContext
- AdditionalLogContext

The eMail sample adapter requires these additional delcarations:

Table 8-13 RA.XML Properties

Property	Example
UserName	The username for eMail adapter login.
Password	The password for username.
ConnectionURL	URL to the eMail server.

You can view the complete `ra.xml` file for the eMail adapter in:

- `*WLAI_HOME\dev\email\src\rar\meta-inf\`

*`WLAI_HOME` is the drive or home directory where BEA WebLogic Application Integration is installed.

Step 5b: Create the .rar File

Class files, logging configuration, and message bundle(s) should be bundled into a file. This `.jar` file and `META-INF/ra.xml` should then be bundled into `.rar` file. The Ant `build.xml` file demonstrates how to properly construct the `.rar` file.

Step 5c: Deploy the .rar File

The `.rar` file should be deployed into the container in the J2EE-compliant application server. The deployment procedure is different on every J2EE-compliant server.

Creating the eMail Adapter Design-Time GUI

The design-time GUI is the user interface that allows the user to create application views, add services and events and deploy the adapter if it is hosted in the BEA WebLogic Application Integration. This section discusses some specific design time issues that were considered during the development of the eMail adapter.

The process of creating the eMail adapter design-time GUI is comprised of the following steps:

- Step 1: Development Considerations
- Step 2: Determine eMail Adapter Screen Flow
- Step 3: Create the Message Bundle
- Step 4: Implementing the Design-Time GUI
- Step 5: Writing Java Server Pages

Step 1: Development Considerations

Some of the important development considerations regarding the design-time GUI for the eMail adapter included:

- Determine the email server that will be supported
- Determine the eMail schema generation
- Determine if the adapter should support testing of service and events.

Step 2: Determine eMail Adapter Screen Flow

You should consider the order in which the Java server pages will appear when the user displays the application view.

Java Server Pages (JSP)

The eMail adapter uses the ADK's Java server pages for a design-time GUI; however, additional JSPs have been added to provide adapter-specific functionality. A description of the additional JSPs follows:

Filename	Description
`addevent.jsp`	The Add Event page allows a user to add a new event to the application view.
`addservc.jsp`	The Add Service page allows the user to add a new service to the application view.
`confconn.jsp`	The Confirm Connection page provides a form for a user to specify connection parameters for the EIS.
`display.jsp`	The Display page is the main "displayer" for other java server pages in the adapter. It authenticates the request and sets the basic look of each screen with a title and description. It includes each JSP that is requested and displays it inline.

Step 3: Create the Message Bundle

To support the Internationalization of all text labels, messages, exceptions, etc., the eMail adapter uses a message bundle based on a text property file. The property file uses copied name value pairs from the BEA_WLS_SAMPLE_ADK property file, and new entries were added for specific to the eMail adapter.

The message bundle for the eMail adapter is contained in `WLAI_HOME\dev\doc\email\src` directory, which was installed with the ADK. The "`HOME`" indicates the drive used during installation. Please refer to `BEA_WLS_email_ADK.properties` in the directory above.

For additional instructions on creating a message bundle, please refer to the JavaSoft tutorial on internationalization at: java.sun.com/docs/books/tutorial/i18n/index.html.

Step 4: Implementing the Design-Time GUI

To implement the design-time GUI, you need to create a DesignTimeRequestHandler class. This class accepts user input from a form and performs a design-time action.

For more information, see "Step 4: Implementing the Design-Time GUI" in "Developing a Design-Time GUI."

eMail Implementation

The Email DesignTimeRequestHandler class extends AbstractDesignTimeRequestHandler and provides these methods:

Method	Description
`addevent(javax.servlet.http.HttpServlet Request request)`	Adds an event to the application view.
`addservc(javax.servlet.http.HttpServlet Request request)`	Adds a service to the application view.
`getAdapterLogicalName()`	Returns my adapter's logical name and helps parent when deploying application views, etc.
getManagedConnectionFactoryClass()	Returns my adapter's SPI ManagedConnectionFactory implementation class, used by parent to get a CCI connection to my EIS.

Step 5: Writing Java Server Pages

Step 5a: Developers' Comments

1. Your Java Server Pages will be displayed within your `control.jsp`; thus `control.jsp` is the first .jsp that you need to copy. Use the display at the `control.jsp` in the example adapters (DBMS and eMail) of the ADK as a starting point.

2. The ADK provides a library of custom `.jsp` tags, which are used extensively throughout the Java server pages of the ADK and eMail adapter. They provide the ability to add validation, to save field values when the user clicks away, and a number of other features.

Note: There are a number of ways to save an object's state when building your adapter using the ADK. The AbstractDesignTimeRequestHandler maintains an ApplicationViewDescriptor of the application view being edited. This is often the best place to save state. Calls to the handler are fast and efficient. You can also ask the AbstractDesignTimeRequestHandler for a Manager Bean, using its convenience methods: getApplicationViewManager(), getSchemaManager(), and getNamespaceManager(), to retrieve information from the repository about an application view. This is more time-consuming but may be necessary on occasion. Since it is a Java Server Page, you can also use the session object, although everything put in the session must explicitly implement the java.io.serializable interface.

Step 5b: Write the WEB-INF/web.xml Web Application Deployment Descriptor

Write the WEB-INF/web.xml Web application deployment descriptor. In most cases, you should use the sample adapter's `web.xml` file as a starting point and modify the necessary components to fit your needs. The code for the `web.xml` file for the eMail adapter is shown below:

Index

D

E

F

G

I

J

L

T

U

V

W

X