Oracle® Retail Service Layer

Programmer's Guide
Release 14.0
E49186-01

December 2013

ORACLE

Oracle® Retail Service Layer Programmer's Guide, Release 14.0

Primary Author: Gloreen Soans

Value-Added Reseller (VAR) Language

Oracle Retail VAR Applications

The following restrictions and provisions only apply to the programs referred to in this section and licensed to you. You acknowledge that the programs may contain third party software (VAR applications) licensed to Oracle. Depending upon your product and its version number, the VAR applications may include:

(i) the **MicroStrategy** Components developed and licensed by MicroStrategy Services Corporation (MicroStrategy) of McLean, Virginia to Oracle and imbedded in the MicroStrategy for Oracle Retail Data Warehouse and MicroStrategy for Oracle Retail Planning & Optimization applications.

(ii) the **Wavelink** component developed and licensed by Wavelink Corporation (Wavelink) of Kirkland, Washington, to Oracle and imbedded in Oracle Retail Mobile Store Inventory Management.

(iii) the software component known as **Access Via™** licensed by Access Via of Seattle, Washington, and imbedded in Oracle Retail Signs and Oracle Retail Labels and Tags.

(iv) the software component known as **Adobe Flex™** licensed by Adobe Systems Incorporated of San Jose, California, and imbedded in Oracle Retail Promotion Planning & Optimization application.

You acknowledge and confirm that Oracle grants you use of only the object code of the VAR Applications. Oracle will not deliver source code to the VAR Applications to you. Notwithstanding any other term or condition of the agreement and this ordering document, you shall not cause or permit alteration of any VAR Applications. For purposes of this section, "alteration" refers to all alterations, translations, upgrades, enhancements, customizations or modifications of all or any portion of the VAR Applications including all reconfigurations, reassembly or reverse assembly, re-engineering or reverse engineering and recompilations or reverse compilations of the VAR Applications or any derivatives of the VAR Applications. You acknowledge that it shall be a breach of the agreement to utilize the relationship, and/or confidential information of the VAR Applications for purposes of competitive discovery.

The VAR Applications contain trade secrets of Oracle and Oracle's licensors and Customer shall not attempt, cause, or permit the alteration, decompilation, reverse engineering, disassembly or other reduction of the VAR Applications to a human perceivable form. Oracle reserves the right to replace, with functional equivalent software, any of the VAR Applications in future releases of the applicable program.

Contents

Send Us Your Comments

Oracle Retail Service Layer Programmer's Guide, Release 14.0

Oracle welcomes customers' comments and suggestions on the quality and usefulness of this document.

Your feedback is important, and helps us to best meet your needs as a user of our products. For example:

- Are the implementation steps correct and complete?
- Did you understand the context of the procedures?
- Did you find any errors in the information?
- Does the structure of the information help you with your tasks?
- Do you need different information or graphics? If so, where, and in what format?
- Are the examples correct? Do you need more examples?

If you find any errors or have any other suggestions for improvement, then please tell us your name, the name of the company who has licensed our products, the title and part number of the documentation and the chapter, section, and page number (if available).

> **Note:** Before sending us your comments, you might like to check that you have the latest version of the document and if any concerns are already addressed. To do this, access the Online Documentation available on the Oracle Technology Network Web site. It contains the most current Documentation Library plus all documents revised or released recently.

Send your comments to us using the electronic mail address: retail-doc_us@oracle.com

Please give your name, address, electronic mail address, and telephone number (optional).

If you need assistance with Oracle software, then please contact your support representative or Oracle Support Services.

If you require training or instruction in using Oracle software, then please contact your Oracle local office and inquire about our Oracle University offerings. A list of Oracle offices is available on our Web site at www.oracle.com.

Preface

The purpose of this manual is to provide a basic understanding of the Oracle Retail Service Layer components, including the flow of a synchronous call between two applications.

Audience

This manual is designed for System Administrators, Developers, and Applications Support personnel.

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

For more information, see the following documents in the Oracle Retail Service Layer Release 14.0 documentation set:

- *Oracle Retail Service Layer Installation Guide*
- *Oracle Retail Service Layer Release Notes*

Customer Support

To contact Oracle Customer Support, access My Oracle Support at the following URL:

https://support.oracle.com

When contacting Customer Support, please provide the following:

- Product version and program/module name
- Functional and technical description of the problem (include business impact)
- Detailed step-by-step instructions to re-create
- Exact error message received
- Screen shots of each step you take

Review Patch Documentation

When you install the application for the first time, you install either a base release (for example, 14.0) or a later patch release (for example, 14.0.1). If you are installing the base release and additional patch and bundled hot fix releases, read the documentation for all releases that have occurred since the base release before you begin installation. Documentation for patch and bundled hot fix releases can contain critical information

related to the base release, as well as information about code changes since the base release.

Improved Process for Oracle Retail Documentation Corrections

To more quickly address critical corrections to Oracle Retail documentation content, Oracle Retail documentation may be republished whenever a critical correction is needed. For critical corrections, the republication of an Oracle Retail document may at times **not** be attached to a numbered software release; instead, the Oracle Retail document will simply be replaced on the Oracle Technology Network Web site, or, in the case of Data Models, to the applicable My Oracle Support Documentation container where they reside.

This process will prevent delays in making critical corrections available to customers. For the customer, it means that before you begin installation, you must verify that you have the most recent version of the Oracle Retail documentation set. Oracle Retail documentation is available on the Oracle Technology Network at the following URL:

`http://www.oracle.com/technetwork/documentation/oracle-retail-100266.html`

An updated version of the applicable Oracle Retail document is indicated by Oracle part number, as well as print date (month and year). An updated version uses the same part number, with a higher-numbered suffix. For example, part number E123456-**02** is an updated version of a document with part number E123456-**01**.

If a more recent version of a document is available, that version supersedes all previous versions.

Oracle Retail Documentation on the Oracle Technology Network

Documentation is packaged with each Oracle Retail product release. Oracle Retail product documentation is also available on the following Web site:
http://www.oracle.com/technology/documentation/oracle_retail.html

(Data Model documents are not available through Oracle Technology Network. These documents are packaged with released code, or you can obtain them through My Oracle Support.)

Documentation should be available on this Web site within a month after a product release.

Conventions

Navigate: This is a navigate statement. It tells you how to get to the start of the procedure and ends with a screen shot of the starting point and the statement "the Window Name window opens."

```
This is a code sample
    It is used to display examples of code
```

1
Introduction

Oracle Retail Service layer (RSL) handles the interface between a client application and a server application. The client application typically runs on a different host from the service. However, RSL allows for the service to be called internally in the same program or Java Virtual Machine as the client without the need for code modification.

All services are defined using the same basic paradigm -- the input and output to the service, if any, is a single set of values. Errors are communicated through Java Exceptions that are thrown by the services. The normal behavior when a service throws an exception is for all database work performed in the service call being rolled back.

RSL works within the Java EE framework. All services are contained within an interface offered by a Stateless Session Bean. To a client application, each service appears to be only a method call.

Some Oracle Retail applications, such as RMS, are implemented in the PL/SQL language, which runs inside of the Oracle Database. RSL uses a generalized conversion process that converts the input java object to a native Oracle Object and any output Oracle Objects to the returned java object from the service. There is a one-to-one correspondence for all fields contained in the Java parameters as in the Oracle Objects used.

A client does not need to know if the business logic is implemented as an Oracle Stored Procedure or in some other language.

2

Technical Architecture

This chapter describes the overall architecture of the Oracle Retail Service Layer. The RSL architecture is built on Java EE Java Enterprise Bean technology. It comprises different architecture layers that perform specific task within the overall workflow of integration between two applications.

RSL provides two different models for service providers. The election of what model to use depends on what type of application to which the "service provider" developer is adding the RSL layer. For applications that follow the Java EE or simple Java architecture, a Java EE model will be a better fit. An Oracle PL/SQL model is a better fit for applications that heavily rely on database business logic, such as Oracle Forms-based applications (RMS).

"Client application" developers do not need to be aware of this distinction. The developer needs only to implement the code that retrieves an instance of the service proxy using the service interface and the Common Components provided ServiceAccessor class and makes the calls using a predefined set of payload objects for argument and return type.

Architecture Layers

The RSL is divided into a series of layers. These layers are as follow:

- The Client Application Layer (CAL). This layer is developed by client application developers. The code for this is dependent on the business processes of the Client.
- The Service Access Layer (SAL). This layer is generated by the Oracle Retail Integration Team. Its purpose is to provide a typed set of interfaces and implementations for accessing services. The SAL is the only set of interfaces that the CAL developer needs to be aware of. Although different implementations of the SAL might be used by the Client application, depending on the local or remote location of the required services, the CAL developer doesn't need to be aware of this.
- The Service Provider Layer (SPL). This layer is implemented by the developers belonging to the service provider or knowledgeable in the service application domain. Each service must implement its corresponding interface as declared in the SAL. For RMS, RWMS or other Oracle Forms applications, each service will be offered via a PL/SQL Stored Procedure and use Oracle Object technology for input and output parameters. For Java EE offered services, input and output parameters will be generated through Value Objects – old fashioned Java beans that implement a defined interface. They consist of "getter", "setter" and "adder" methods. See the following sections for a discussion of the Java EE and Oracle PL/SQL-based models.

> **Note:** There is a one-to-one mapping of APIs described in the SPL rather than APIs offered in the SAL. For Oracle based applications, a generic Stored Procedure Caller class (provided by the Integration Team) is accessible through the Common Components CommandExecutionServiceEjb Stateless Session Bean. This class handles all RMS provided simple services.

Note: The services offered by a single service should be a logical unit that is functionally cohesive. This has implications if a retailer wishes to use a different implementation, such as a completely home-grown implementation for this functionality.

Java EE Application Model

Figure 1 illustrates the model and workflow to be used when integrating RSL with a Java EE or simple Java application. The objects on the left side of the dashed lines symbolize the client view of the transaction; the right side characterizes the service provider part.

In the diagram, objects in blue are implementations provided by either the client application developer (left side) or service provider developer (right side). The red objects indicate the interfaces, classes, and payload objects provided by the Oracle Retail Integration Team to both developers. Oracle Retail's Common Components objects are denoted in green.

The SPL developer needs to create a POJO (Plain Old Java Object) class that implements the SAL interface provided by the Integration Team. This class should be made available in the Java EE environment through the CommandExecutionService EJB. See Chapter 4, "Service Provider How-To Guide," for an in-depth discussion on how to develop the Service Provider Layer and make it available to RSL client applications.

The CAL developer will create the code that will make it possible for the client application to contact the RSL service. This involves using the Common Components provided ServiceAccessor class to retrieve an instance of the Service Proxy to communicate with the service.

Once this proxy is obtained, the CAL invokes calls as declared in the service interface provided by the Integration Team as part of the SAL deliverables. See Chapter 5, "Client How-To Guide," for how to develop the client code and configurations to successfully invoke RSL services.

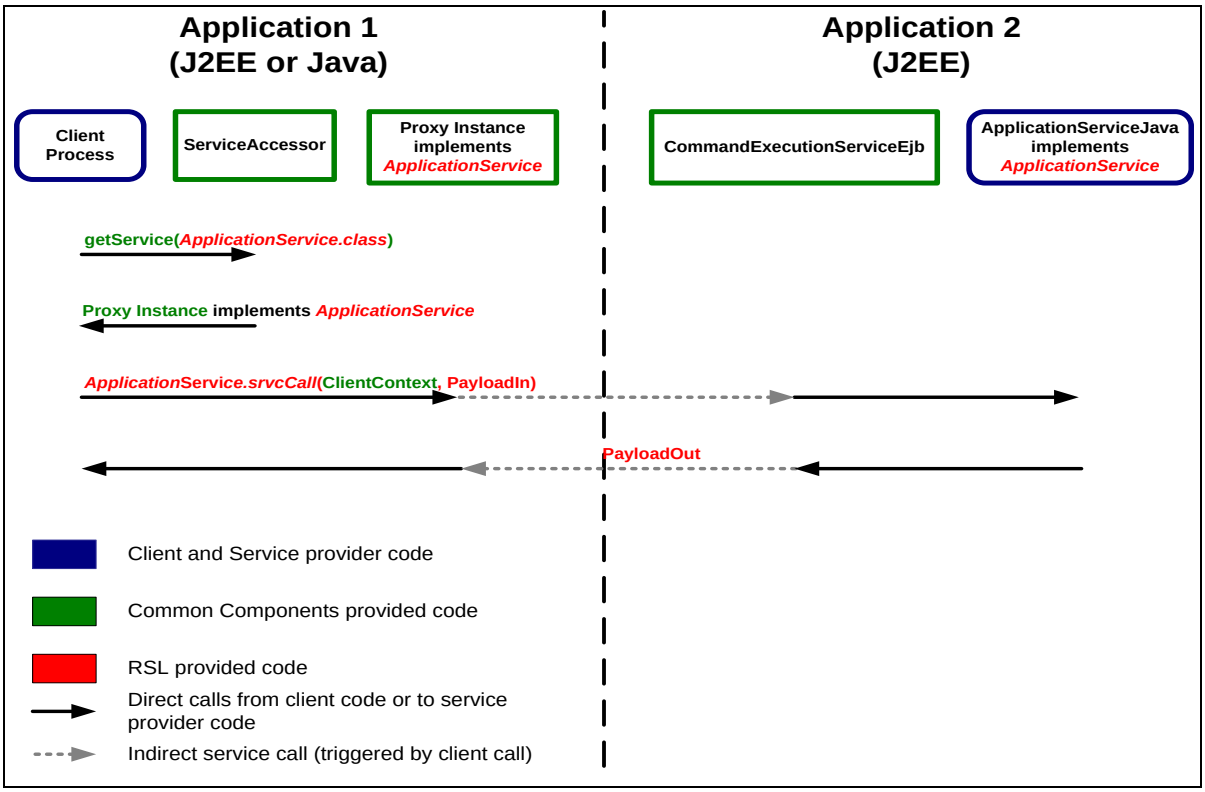

Figure 1

Oracle PL/SQL-based Application Model

Figure 2 shows the model and workflow to be used when integrating RSL with an application based on Oracle PL/SQL language (that is, RMS). The objects on the far left symbolize the client view of the transaction; the objects between both dashed lines characterize the service provider part; finally, the objects on the extreme right represent the Oracle-based application. In this model, the Integration Team is responsible for distributing the "service provider" implementation of RSL. In the diagram, objects in blue are implementations provided by either the "client application" developer (left side) or "Application Team" developer (right side). The red objects indicate the interfaces, classes, and payload objects provided by the Integration Team to both developers. Oracle Retail's Common Components objects are denoted in green.

The Oracle Retail Application Team is responsible for developing the PL/SQL Stored Procedure following the guidelines provided by the Integration Team and discussed in Chapter 4, "Service Provider How-To Guide."

The CAL developer creates the code that allows the client application to contact the RSL service. This involves using the Common Components provided ServiceAccessor class to retrieve an instance of the Service Proxy to communicate with the service. Once this proxy is obtained, the CAL invokes calls as declared in the service interface provided by the Integration Team as part of the SAL deliverables. See Chapter 5, "Client How-To Guide," for how to develop the client code and configurations to successfully invoke RSL services.

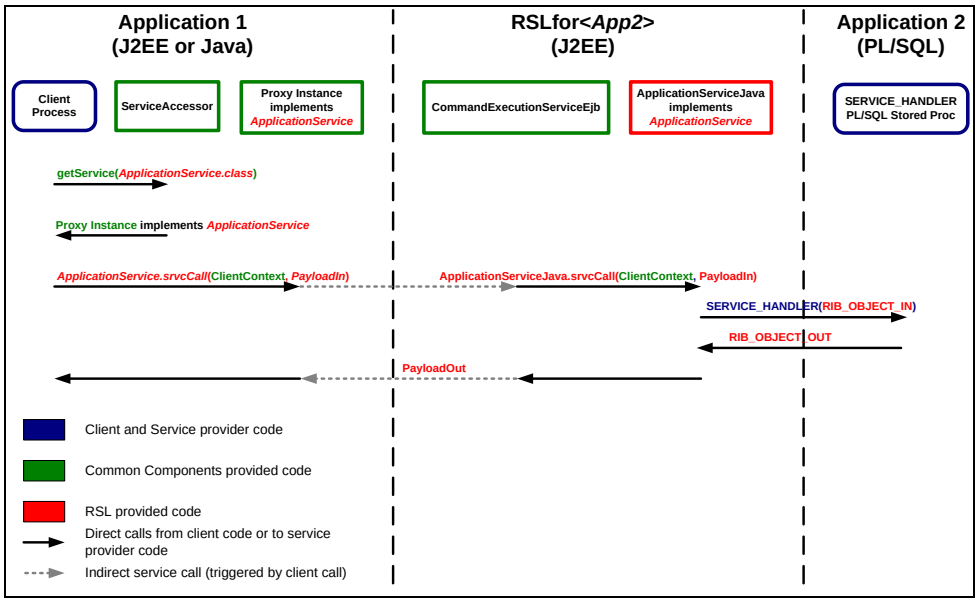
Application 1
(J2EE or Java)
RSLfor<App2>
(J2EE)
Application 2
(PL/SQL)
Client Process
ServiceAccessor
Proxy Instance implements ApplicationService
CommandExecutionServiceEjb
ApplicationServiceJava implements ApplicationService
SERVICE_HANDLER PL/SQL Stored Proc
getService(ApplicationService.class)
Proxy Instance implements ApplicationService
ApplicationService.srvcCall(ClientContext, PayloadIn)
ApplicationServiceJava.srvcCall(ClientContext, PayloadIn)
SERVICE_HANDLER(RIB_OBJECT_IN)
RIB_OBJECT_OUT
PayloadOut
Client and Service provider code
Common Components provided code
RSL provided code
Direct calls from client code or to service provider code
Indirect service call (triggered by client call)

Figure 2

3

Basic Operations

This section details the synchronous data flow of a payload between end applications. The business logic is completely owned by the application implementing the service.

Simple Service Call

In the client application, some event triggers a service call. The first step is for the client to find the location of the service. This is done by using the ServiceAccessor, which connects to the application server's Java Naming and Directory Interface (JNDI) instance. Once the Remote Home of the service is located, the client creates a Remote instance (or handle) to the service, which is returned to the client to perform calls to the service.

At this point, the Java EE application server infrastructure takes control and performs a remote method invocation on the CommandExecutionService Stateless Session Bean. This EJB is responsible for looking up the implementation of the service interface, invoking the client requested service method call and returning the result.

The execution and control of database commits and rollbacks is dependent on three factors: the configuration of the Stateless Session Bean, the success or failure of the service call, and whether the transaction is started by the client or the application server.

Configuration of the Stateless Session Beans

Normally, RSL Beans are configured with Container Managed transactions. This means that the Application Server EJB Container decides whether database work will be committed or not. Furthermore, there is the assumption that the configuration of the database connection is a "Container Managed" resource. Within a container, each resource has a specific name. A service may use one or more resources during its execution.

Success or Failure of the Service Call

If an error is encountered during the service execution, normal behavior is to roll back all database work. This is performed when an exception is thrown by the service for container managed transactions.

Clients Starting a Transaction

Transactions for most service calls are started by the service implementation or the Application Server, not by the client. However, it is possible for the client to start a transaction and make multiple service calls within the same transaction.

4

Service Provider How-To Guide

This chapter provides information about how to configure the service provider application.

Service Provider Application Configuration

As mentioned earlier, the Service itself is developed by the service provider application. The following files need to be properly configured in the service provider application.

retek/services_rsl.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <customizations>
        <interface package="com.retek.rsl.<app>">
            <impl package="<application specific>" />
        </interface>
    </customizations>
</services-config>
```

The name of the package for the interface usually follows the `com.retek.rsl.<app>` convention, where *<app>* is substituted by the name of the service provider application, such as rms and rpm.

The implementation package name is provided by the service provider once the code is developed; this name is irrelevant for the client application.

retek/service_flavors.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <flavors set="server">
        <flavor name="java"
locator="com.retek.platform.service.SimpleServiceLocator" suffix="Java"/>
    </flavors>
</services-config>
```

Service Provider Application Layer Code for J2EE Models

An example of a SPL Service implementation is shown below. The sample implements the PriceInquiryService interface. All methods have been stubbed out, because the logic is unknown to anyone outside of the service providing application.

> **Note:** The name of these implementation classes should follow a naming convention, such as ServiceInterfaceNameJava (see example above). This allows the Common Components service framework to locate the SPL's implementation class by using the "server" flavor. This naming convention is configured in the SPL's service_flavors.xml file.

```
package com.retek.rpm.service;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import com.retek.platform.exception.RetekBusinessException;
import com.retek.platform.service.ClientContext;
import com.retek.platform.service.FallbackHandler;
import com.retek.rib.binding.payload.PrcInqDesc;
import com.retek.rib.binding.payload.PrcInqReq;
import com.retek.rsl.rpm.PriceInquiryService;

public class PriceInquiryServiceJava implements PriceInquiryService {

    protected final Log LOG = LogFactory.getLog(getClass());

    public void attachFallbackHandler(FallbackHandler arg0) {
LOG.debug("Executing business logic for attachFallbackHandler");
    }

public PrcInqDesc prcinqqry(ClientContext client, PrcInqReq query) throws
RetekBusinessException {
        LOG.debug("Executing business logic for query");
        return new PrcInqDesc();
    }

}
```

Service Provider Application Layer Code for Oracle PL/SQL-Based Models

For each service interface in the Service Access Layer, the application developer must create a package that contains a stored procedure named SERVICE_HANDLER. The signature of this stored procedure is as follow:

```
PROCEDURE SERVICE_HANDLER(O_status      OUT    RIB_OBJECT,
                          O_payload     OUT    RIB_OBJECT,
                          I_action      IN     VARCHAR2,
                          I_payload     IN     RIB_OBJECT,
                          I_client      IN     RIB_OBJECT);
```

The O_status return object should be an instance of a RIB_STATUS_REC Oracle type object in the database. This object contains two variables. A status_code variable of type varchar2 that holds the status of the SERVICE_HANDLER() call; a value of S indicates that the call was successful. Any other status code should be accompanied by a description, assigned to the second varchar2 variable of the RIB_STATUS_REC object.

The O_payload return object is the value returned to the client application after the service call.

I_action is a varchar2 representing what type of action to perform for the given payload. Actions should match one-to-one to methods in the service interface. Each method call from the client application passes a different I_action value to the SERVICE_HANDLER() stored procedure. In this way, the application developer can route the request to different business processes in their application.

I_payload corresponds to the object sent by the client application and used by the stored procedure to perform some business process action.

I_client is an object of type RIB_CLIENT_REC that represents the instance of the ClientContext object sent by the client application to the RSL service. This ClientContext is translated to a RIB_CLIENT_REC Oracle type that can be used by the application developer to identify the context used for the invocation of this call.

5

Client How-To Guide

This chapter provides information about how to configure the client application layer.

Client Application Layer Configuration

The Client Application Layer Configuration (CAL) is developed by the "client application" developer. The following files must be properly configured in the client application.

retek/services_rsl.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <customizations>
        <interface package="com.retek.rsl.<app>" app="<app>">
            <impl package="" />
        </interface>
    </customizations>
</services-config>
```

An entry like the following should exist in the client application's services_rsl.xml file. The name of the package for the interface usually follows the "`com.retek.rsl.`*<app>*" convention, where *<app>* is substituted by the name of the service provider application, such as rms and rpm.

The implementation package name is irrelevant in the client application, since this will go through the CommandExecutionService EJB to make the service calls.

retek/service_flavors.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <flavors set="client">
        <flavor name="ejb" locator="com.retek.platform.service.EjbServiceLocator"
remote-suffix="Remote" home-suffix="RemoteHome"/>
    </flavors>
</services-config>
```

retek/jndi_providers.xml

```
<?xml version="1.0" ?>
<ejb_context_overrides>
        <provider app="rms" map-name="rsl-rms" csm-wallet="../conf">
                <encrypted-context-property name="java.naming.security.principal"
key="rms-alias" value-type="PC_USERNAME" />
                <encrypted-context-property
name="java.naming.security.credentials" key="rms-alias" value-type="PC_PASSWORD"
/>
                <context-property name="java.naming.provider.url"
value="t3://<host>:<port>" />
                <context-property name="java.naming.factory.initial"
value="weblogic.jndi.WLInitialContextFactory" />
        </provider>
</ejb_context_overrides>
```

Client Application Layer Code

An example of a CAL method call is below. This sample calls the "prcinqqry" method of PriceInquiryService. The Service interface and the input and output Payload classes are provided by the Integration Team.

One parameter shown below is the Oracle Retail Common Components' ClientContext object. The CAL must create this object. It is used for application logging and tracking purposes. It is a required object on every service interface. Other than the constructor parameters, this object identifies the name of the host on which the object was created, the Process ID, the Thread ID that created the object, and the Locale (or language) of the client.

```
import com.retek.platform.exception.RetekBusinessException;
import com.retek.platform.service.ClientContext;
import com.retek.platform.service.ServiceAccessor;
import com.retek.platform.util.type.security.SecureUser;
import com.retek.rib.binding.payload.PrcInqDesc;
import com.retek.rib.binding.payload.PrcInqReq;

public class ClientCode {
    public PrcInqDesc prcinqqry(PrcInqReq request)
throws RetekBusinessException {

// create a client ID. This may be cached and reused.
              ClientContext ctx = ClientContext.getInstance();
SecureUser user = new SecureUser("username", "firstname",
    "lastname", null);
      ctx.setUser(user);
              ctx.setLocale(Locale.US);

// create a new client handle.
PriceInquiryService service =
ServiceAccessor.getService(PriceInquiryService.class);

// call the query method on service
try {
PrcInqDesc price = service.prcinqqry(request);
return price;
} catch  (RetekBusinessException rbe) {
    throw rbe;
}
    }
}
```

6

RSLTestClient Utility

The RSL Client Test Application is a tool that helps verify that a Client Application can successfully establish communication with an RSL Service Provider Application.

The test consists of very simple service calls that return a message indicating that a call has been made with a specific timestamp. In case of errors, it displays a message indicating the possible causes for the problem and how to fix it.

This test supports the RSL's Java EE Application Model and the Oracle PL/SQL-based Application Model. If the RSL Service Provider follows the Java EE Application Model, an error message might occur during execution of the last test. As indicated by the error, this is not a concern if the test fails in the Java EE Application Model, as the last test might not be applicable for such model. If the RSL Service Provider follows the Oracle PL/SQL-based Application Model, be sure that all installations instructions have been followed as outlined below.

Installation and Configuration

The first two steps are required only if the RSL Service Providers follow the Oracle PL/SQL-based Application Model; if they do not, skip these steps and go directly to Step 3.

1. Load the RSL Test objects in the database by executing the `@CreateRslTestObjects` command in a SQL Plus terminal. The CreateRslTestObjects.sql script should be found in the testapp directory of the RSL Server or Client pak.
2. Load the RSLSVC_TEST package in the database. This can be done by executing the `@RSLSVC_TEST.pkb` command in a SQL Plus terminal. The RSLSVC_TEST.pkb script should be found in the testapp directory of the RSL Server or Client pak.
3. Edit the value of the property, <context-property name="java.naming.provider.url," in the jndi.properties file, located in the conf/retek directory of the RSL Client pak to point to the host and port where the RSL server has been deployed.

Execution

Make sure the RSL Service Provider Application is up and running. Execute the rsltestclient.sh script. It will prompt for the username/password for the WebLogic server where the RSL service provider application is running. After all tests have been completed, the following message is displayed to indicate that all tests completed successfully:

```
>>>>> Test completed successfully <<<<<
```

Or the following message may be displayed to indicate that errors may have occurred during the execution of one of the tests:

```
>>>>> Test completed with possible errors, see sections above <<<<<
```

Once an error has been detected, the other tests will not be performed. Examine the output to determine the cause of the error.

When executing this test against an RSL Service Provider that follows the Java EE Application Model, and if steps 1 and 2 in the "Installation and Configuration" section were not followed, the following message is displayed, even if all tests passed successfully:

```
>>>>> Test completed with possible errors, see sections above <<<<<
```

If the last test that ran was the SERVICE_HANDLER stored procedure test, ignore this message and assume all tests passed successfully; because this is the last test, and it is not relevant for Java EE Application Models.

A

Appendix: Configuration Files

The RSL uses a variety of configuration files. These are mostly related to Common Components packaged as part of RSL to allow independent configuration.

Services Framework Configuration

The following XML files are part of the services framework configuration.

jndi_providers.xml

The file must be in the classpath located in a *retek* directory. The JNDI providers's XML is used by the EJBServiceLocator to lookup Services in remote JNDIs. See the section, "Common Components," for specific formatting of this file.

services_rsl.xml

The file must be in the classpath located in a *retek* directory. Within this file are the package locations of the Service Implementations, which are used to configure the ServiceFactory. See the section, "Common Components," for specific formatting of this file.

service_flavors.xml

The file must be in the classpath located in a *retek* directory. Within this file are flavorsets, which are used to configure the ServiceFactory. See the section, "Common Components," for specific formatting of this file.

service_context_factory.xml

The file must be in the classpath located in a *retek* directory. This specifies the factory class used by Common Components to retrieve a service context implementation for RSL.

JDK Logging Configuration

RSL utilizes the commons-logging framework provided by Jakarta.apache.org. The current property file is configured to use JDK logger as its default factory.

J2ee-logging

This file must be in the classpath. This is the main configuration file for JDK logging.

Other Configuration

commons-logging.properties

This file must be in the classpath. It contains the correct logging factory to instantiate for RSL. We utilize the Jdk14Logger, but you may configure you application differently through this file.

B

Appendix: Common Libraries

This section includes the third party jars necessary for RSL functionality. It also describes why each is necessary and what it provides to the framework.

RSL is built upon multiple existing technologies developed within Oracle Retail. Its reuse is essential to interoperability between the RSL and other applications.

Common Components

The following .jar files are common components.

platform-server.jar

This jar contains common components specific code for Oracle Retail. This includes the Service framework. All services developed will be dependant on this jar for its use of AbstractService and Service related classes.

platform-api.jar

This jar contains common components specific code for Oracle Retail. This includes Exceptions and other Objects that could be transferred "over the wire".

platform-conf.jar

This jar contains common components specific configuration files.

platform-resources.jar

This jar contains common components specific resource files for Oracle Retail.

platform-common.jar

This jar contains common components specific code for Oracle Retail.

RSL/Integration

oo-jaxb-bo-converter.jar

This jar contains specific code for Oracle Retail to convert Oracle Objects to Java and vice versa.

rib-public-payload-java-beans.jar

This jar contains the java bean representation of business objects. They are collectively referred to as payloads. The payloads are generated utilizing the RIB artifact generator. They are dependant on castor for marshalling and unmarshalling. Payloads located in this jar are type specific, this may cause interoperability issues with legacy applications that are non-type specific, but follows in the future path of the project.

rib-public-api.jar

This jar contains external public RIB APIs. Java interfaces, factories and exception reside in this jar.

rib-private-kernel.jar

This jar contains bulk of the RIB kernel code. This jar contains various utilities for use with Oracle Retail projects. Included is an OracleStructDumper, which aides in the debugging of converting objects. There are also various string and factory objects that may be of use.

rsl.jar

This jar contains utility, helper and exception classes used by the RSL framework.

rsl-<service provider application>-access.jar

This jar contains the specific interfaces to access Services from another application (SPL). These classes will ultimately be automatically generated by the RIB Framework GUI.

rsl-<service provider application>-business-logic.jar

This jar contains specific RSL implementations of the interfaces. The CommandExcecutionServiceEjb will look up these Implementations. This jar is provided only for Oracle PL/SQL-based applications where the Integration Team provides the Service Provider Layer.

Third Party Provided

Most of these jars are provided via open source. It is recommended to use the packaged jars within the RSL ear and not upgrades. The RSL will provide updates in related releases that will include the update to these jars.

castor-1.0.5-xml.jar

This jar contains classes related to the castor subsystem. Payloads for marshalling and un-marshalling between XML and Java Bean representations utilize it. Documentation and related information is located here: http://castor.exolab.org

ojdbc6.jar

This jar contains classes related to the Oracle JDBC driver. It is utilized within the conversion utilities in the rsl.jar. This jar file has been updated for JDK 1.5

commons-lang.jar

This jar contains various utility classes for use in math, string and time operations. Documentation and related information is located here:
http://jakarta.apache.org/commons/lang/

dom4j.jar

This jar contains XML DOM processing ability. It is needed for processing of the configuration file in xml format. Documentation and related information is located here:
http://www.dom4j.org/

commons-collections.jar

This jar contains various utility classes for use with the Java collection API. Documentation and related information is located here: http://jakarta.apache.org/commons/collections/

commons-logging.jar

This jar contains various utility classes for use with Java Logging. It provides an additional abstraction layer to common logging functionality. Documentation and related information is located here: http://jakarta.apache.org/commons/logging/