Oracle® Retail Service Layer Programmer's Guide
Release 12.0
May 2006

ORACLE

Contents

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.

Related Documents

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

Customer Support

- https://metalink.oracle.com

When contacting Customer Support, please provide:

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

Third-Party Open-Source Applications

Oracle Retail Security Manager includes the following third-party open-source applications:

Software Provider: Intalio Inc.
Software Name: Castor
Software Version: 0.9.5.2
Jar File Name: castor-0.9.5.2.jar
Provider Web Site: www.castor.org
License: ExoLab Public License (based on the BSD License)

Software Provider: The Apache Software Foundation
Software Name: Commons Beanutils
Software Version: 1.6
Jar File Name: commons-beanutils.jar
Provider Web Site: http://jakarta.apache.org/commons/beanutils/
License: The Apache Software License, Version 1.1

Software Provider: The Apache Software Foundation
Software Name: Commons Collections
Software Version: 2.1
Jar File Name: commons-collections.jar
Provider Web Site: http://jakarta.apache.org/commons/collections/
License: The Apache Software License, Version 1.1

Software Provider: The Apache Software Foundation
Software Name: Commons Digester
Software Version: 1.5
Jar File Name: commons-digester.jar
Provider Web Site: http://jakarta.apache.org/commons/digester/
License: The Apache Software License, Version 1.1

Software Provider: The Apache Software Foundation
Software Name: Commons Lang
Software Version: 2.0
Jar File Name: commons-lang.jar
Provider Web Site: http://jakarta.apache.org/commons/lang/
License: Apache License Version 2.0

Software Provider: The Apache Software Foundation
Software Name: Commons Logging
Software Version: 1.0.3
Jar File Name: commons-logging.jar
Provider Web Site: http://jakarta.apache.org/commons/logging/
License: The Apache Software License, Version 1.1

Software Provider: The Apache Software Foundation
Software Name: Commons Validator
Software Version: 1.0.2
Jar File Name: commons-validator.jar
Provider Web Site: http://jakarta.apache.org/commons/validator/
License: The Apache Software License, Version 1.1

Software Provider: dom4j.org
Software Name: DOM4J Project
Software Version: 1.4
Jar File Name: dom4j.jar
Provider Web Site: http://www.dom4j.org
License: BSD style license

Software Provider: The Apache Software Foundation
Software Name: Apache Jakarta Project
Software Version: 1.3
Jar File Name: jakarta-regexp.jar
Provider Web Site: http://jakarta.apache.org/regexp/
License: The Apache Software License, Version 1.1

Software Provider: JDOM Project
Software Name: jdom
Software Version: 1.0 beta9
Jar File Name: jdom.jar
Provider Web Site: http://www.jdom.org/
License: Apache: BSD/Apache style

Software Provider: JUnit.org
Software Name: JUnit
Software Version: 3.8.1
Jar File Name: junit.jar
Provider Web Site: http://www.junit.org/index.htm
License: Common Public License Version 1.0

Software Provider: The Apache Software Foundation
Software Name: Log4j
Software Version: 1.2.6
Jar File Name: log4j.jar
Provider Web Site: http://logging.apache.org/log4j/docs/index.html
License: The Apache Software License, Version 1.1

Software Provider: The Apache Software Foundation
Software Name: Xerces
Software Version: 2.0.2
Jar File Name: xercesImpl.jar
Provider Web Site: http://xerces.apache.org/
License: The Apache Software License, Version 1.1

Software Provider: The Apache Software Foundation
Software Name: xml-commons
Software Version: 1.0.b2
Jar File Name: xml-apis.jar
Provider Web Site: http://xml.apache.org/commons/
License: The Apache Software License, Version 1.1

1
Introduction

RSL Overview

RSL handles the interface between a client application and a server application. The client application typically runs on a different host than 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 via 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 J2EE framework. All services are contained within an interface offered by a Stateless Session Bean. To a client application, each service appears to be merely 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 of 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

Overview

This chapter describes the overall architecture of the Oracle Retail Service Layer. The RSL architecture is built upon J2EE's Java Enterprise Bean technology. It is composed of 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 the "service provider" developer is adding the RSL layer to. For applications that follow the J2EE or simple Java architecture, a J2EE model will be a better fit. An Oracle PL/SQL model is better fit for applications that heavily depend 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 only needs 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:

- 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, RDM 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 J2EE offered services, input and output parameters will be generated via Value Objects – old fashioned java beans that a) implement a defined interface and b) consist of "getter", "setter" and "adder" methods. Refer to the next sections for a discussion of the J2EE and Oracle PL/SQL-based models.

> **Note:** There is a one-to-one mapping of APIs detailed in the SPL versus APIs offered in the SAL. For Oracle based applications, a generic "Stored Procedure Caller" class, provided by the Integration Team, will be accessible through the Common Components CommandExecutionServiceEjb Stateless Session Bean. This class will handle 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.

J2EE Application Model

Figure 1 depicts the model and workflow to be used when integrating RSL with a J2EE or simple Java application. The objects on the left side of the dashed lines symbolize the client view of the transaction while 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 J2EE environment through the CommandExecutionService EJB. Please refer to 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. Chapter 5 – Client How-To Guide explains in details how to develop the client code and configurations to successfully invoke RSL services.

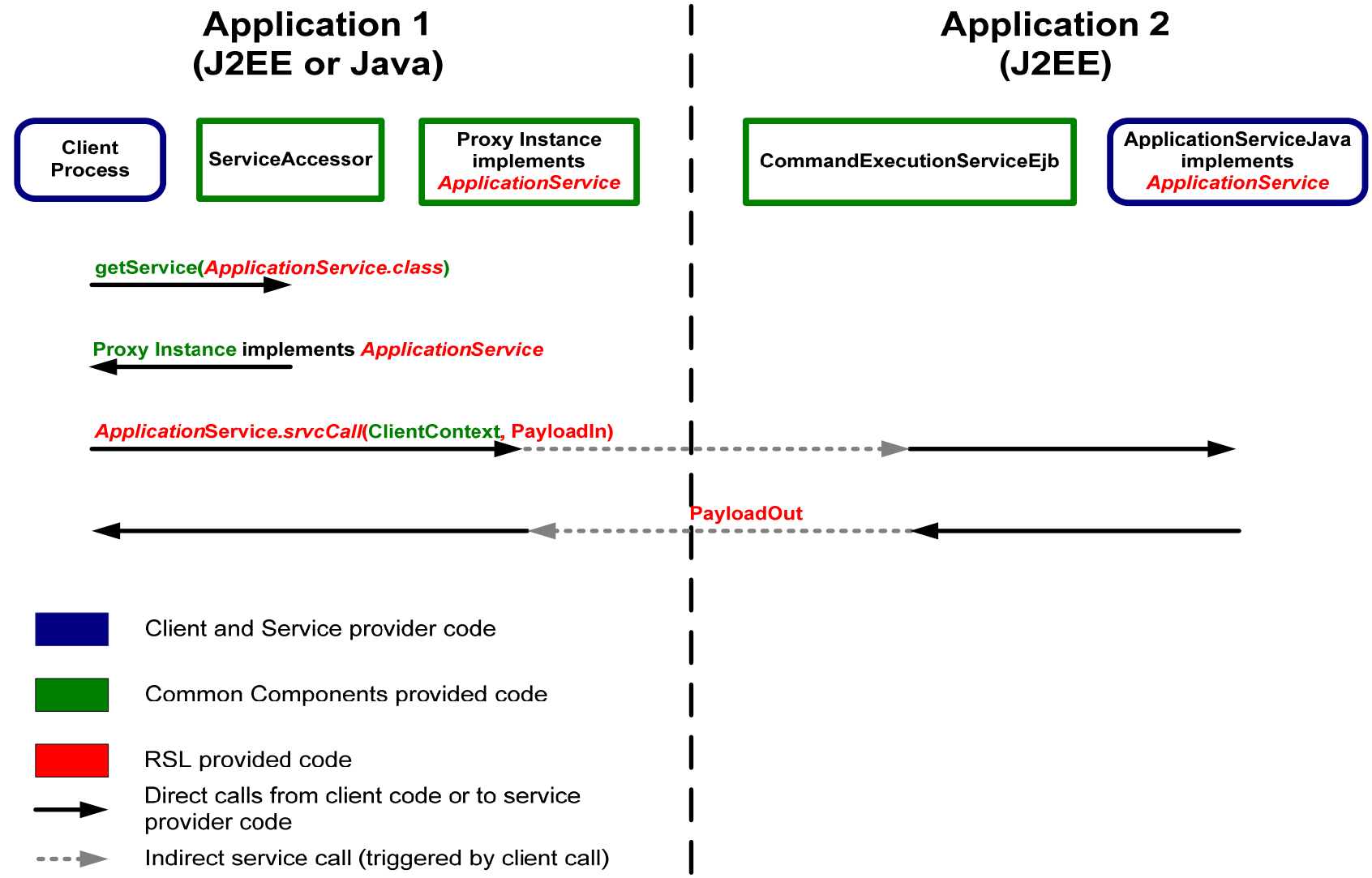
Application 1
(J2EE or Java)
Application 2
(J2EE)
Client Process
ServiceAccessor
Proxy Instance implements ApplicationService
CommandExecutionServiceEjb
ApplicationServiceJava implements ApplicationService
getService(ApplicationService.class)
Proxy Instance implements ApplicationService
ApplicationService.srvcCall(ClientContext, PayloadIn)
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 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 details in Chapter 4 – "Service Provider" How-To Guide.

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. Chapter 5 – Client How-To Guide explains in details how to develop the client code and configurations to successfully invoke RSL services.

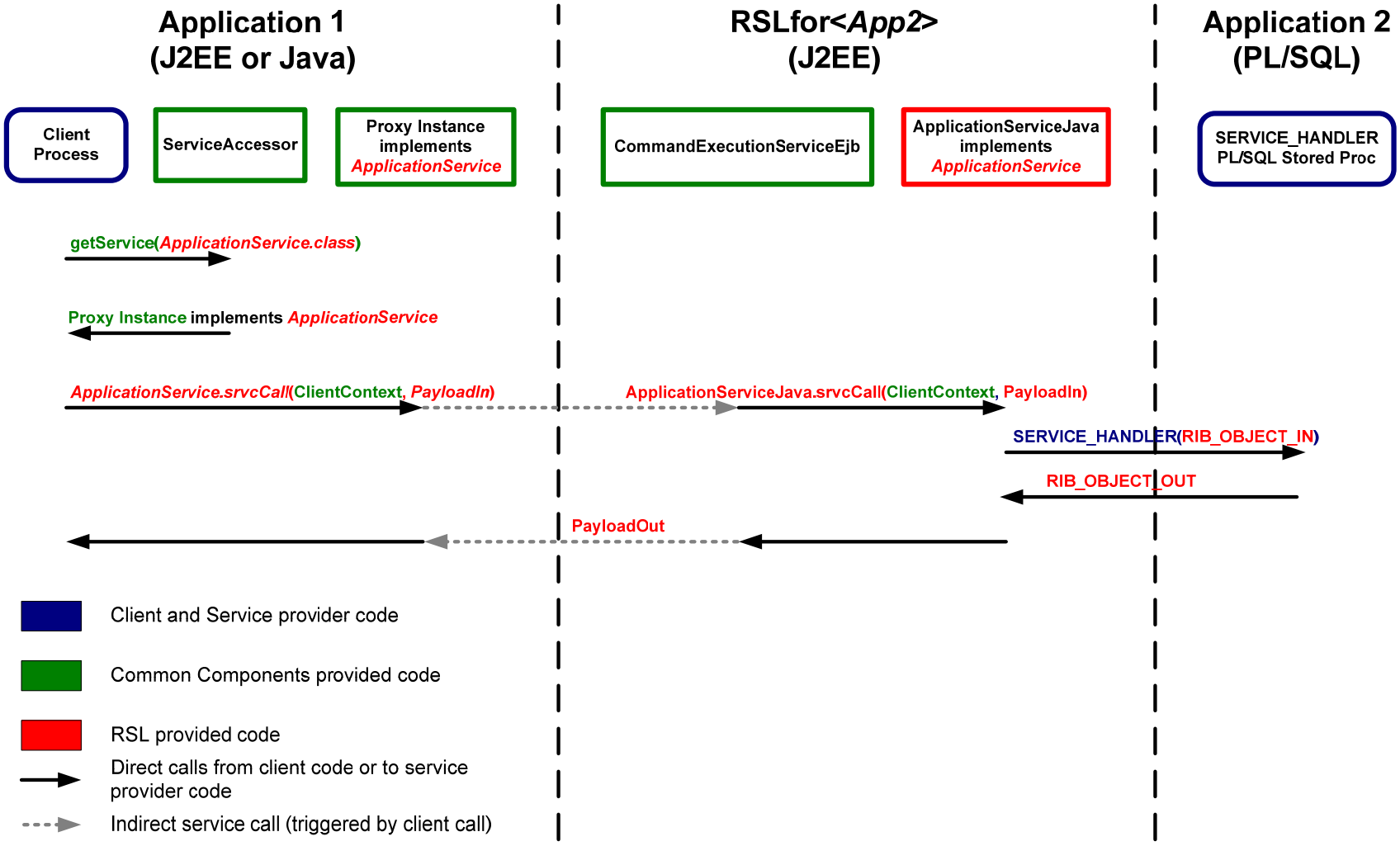
Application 1
(J2EE or Java)
RSLfor<App2>
(J2EE)
Application 2
(PL/SQL)
Client Process
ServiceAccessor
Proxy Instance implements ApplicationService
CommandExecutionServiceEjb
ApplicationServiceJava implements ApplicationService
SERVICE_HANDLER PL/SQL Stored Proc
getService(ApplicationService.class)
Proxy 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 then 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 J2EE 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 will decide if 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: Most service calls have their transactions started by the service implementation or the Application Server and 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

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*> will be substituted by the name of the service provider application: rsm, rpm.

The implementation package name will be 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 seen below. This sample implements the PriceInquiryService interface. All of the methods have been stubbed out, as 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” will indicate 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 that will be 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 will pass a different I_action value to the SERVICE_HANDLER() stored procedure; 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

Client Application Layer Configuration

As mentioned earlier, the CAL is developed by the “client application” developer. The following files need to 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*> will be substituted by the name of the service provider application: rsm, 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>
    <!-- For managed OC4J -->
    <provider app="<app>"
url="opmn:ormi://<oas_host>:<oas_port>:<app_oc4j_instance_name>/<app>"
factory="oracle.j2ee.rmi.RMIInitialContextFactory" />

    <!-- For standalone OC4J -->
<provider app="<app>"
url="ormi://<oas_host>:<oas_port>/<app>"
factory="oracle.j2ee.rmi.RMIInitialContextFactory" />
</ejb_context_overrides>
```

Client Application Layer Code

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

One parameter seen 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 will be a required object on every service interface. Besides the constructor parameters, this object will identify the host name the object was created on, 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 created to help verify that a Client Application can successfully establish communication with an RSL Service Provider Application. This tool is compatible with RSL releases 11.1.0 or newer.

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

This test support both RSL's J2EE Application Model and Oracle PL/SQL-based Application Model. If the RSL Service Provider follows the J2EE 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 J2EE 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, then make sure that all installations instructions have been followed as outlined by the next section.

Installation and Configuration

The first two steps are only required if the RSL Service Providers follows the Oracle PL/SQL-based Application Model; if not, please ignore these steps and go directly to step 3.

1. Load the RSL Test objects in the database. This can be done 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 jndi.properties file located in the testapp directory of the RSL Server or Client pak and enter the java.naming.security.principal and java.naming.security.credentials values. This must be the same as the username and password of the user that has started the OC4J instance. Typically an administrative account created when a new OC4J instance was created.

4. Edit the rsltestclient.bat or rsltestclient.sh script and customize it for the Client Application environment:
   a. Set the JAVA_HOME variable to the location of a 1.4.1 or higher JDK installation.
   b. Set the RSL_SERVER_PROVIDER variable to OC4J.
   c. The CONFIG_DIR variable should be set to the location of the directory that contains the retek/jndi_providers.xml, retek/service_flavors.xml and retek/services_rsl.xml files to be used by the Client Application. Do not include "retek" in the path. For this test, it is recommended the use of the same configuration files as the Client Application will use when released.
   d. Set CLIENT_LIB to the folder that contains all the jar files required for this test. The RSL paks do not distribute these jar files (except for the RSL specifics.) It is assumed that all jar files will be already available in the Client Application library directory.

Execution

Make sure the RSL Service Provider Application is up and running; then execute the rsltestclient.bat or rsltestclient.sh script. After all tests have been completed, the following message will be shown:

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

indicating that all tests completed successfully, or the following message:

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

indicating that there might have been an error during the execution of one of the tests. Once an error has been detected, the other tests will not be performed. Please examine the output to determine the cause of the error.

When executing this test against an RSL Service Provider that follows the J2EE Application Model, and if steps (1) and (2) in the “Installation and Configuration” section were not followed, the following message will be shown

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

even if all tests passed successfully. Ignore this message and assume all tests passed successfully if the last test ran was the SERVICE_HANDLER stored procedure test, as this is the last test and is not relevant for J2EE Application Models.

A

Appendix: Configuration Files

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

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 JNDI's. Please reference the Common Components documentation 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. Please reference the Common Components documentation 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. Please reference the Common Components documentation 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.

Log4j Configuration

We utilize the commons-logging framework provided by Jakarta.apache.org. The current property file is configured to use Log4J as its default factory.

log4j.dtd

This file must be in the classpath. This is required for validation of the log4j.xml. For more information regarding the use of this file, please review
http://logging.apache.org/log4j/docs/

log4j.xml

This file must be in the classpath. This is the main configuration for log4j. For more information regarding the use of this file, please review
http://logging.apache.org/log4j/docs/

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 Log4Jfactory, but you may configure you application differently via this file.

B

Appendix: Common Libraries

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

The 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

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-common.jar

This jar contains common components specific code for Oracle Retail.

RSL/Integration

retek-payload-typed.jar

This jar contains the java bean representation of business objects. They are collectively referred to as payloads. The payloads are generated utilizing the data in the RIB Framework Gui. 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.

retek-rib-support.jar

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 Interface(s) 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 Interface(s). 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-0.9.5.2.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 at http://castor.exolab.org

ojdbc14.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 1.4. Documentation and related information is located at http://otn.oracle.com/software/tech/java/sqlj_jdbc/htdocs/jdbc9201.html

commons-lang-2.0.jar

This jar contains various utility classes for use in math, string and time operations. Documentation and related information is located at 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 at http://www.dom4j.org/

commons-collections-3.1.jar

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

commons-logging.jar and commons-logging-api.jar

These jar contain various utility classes for use with Java Logging. They provide an additional abstraction layer to common logging functionality. Documentation and related information is located at http://jakarta.apache.org/commons/logging/

log4j.jar

This jar contains the log4j subsystem. This is the RSL's logging system of choice and is utilized through the commons-logging.jar. Documentation and related information is located at http://logging.apache.org/log4j/docs/index.html.

Other jar files included with the RSL release are required by the Oracle Retail's Common Components.