

Retek[®] Service Layer[™] 11.0.2

Programmer's Guide

Corporate Headquarters:

Retek Inc.
Retek on the Mall
950 Nicollet Mall
Minneapolis, MN 55403
USA
888.61.RETEK (toll free US)
Switchboard:
+1 612 587 5000
Fax:
+1 612 587 5100

European Headquarters:

Retek
110 Wigmore Street
London
W1U 3RW
United Kingdom
Switchboard:
+44 (0)20 7563 4600
Sales Enquiries:
+44 (0)20 7563 46 46
Fax:
+44 (0)20 7563 46 10

The software described in this documentation is furnished under a license agreement, is the confidential information of Retek Inc., and may be used only in accordance with the terms of the agreement.

No part of this documentation may be reproduced or transmitted in any form or by any means without the express written permission of Retek Inc., Retek on the Mall, 950 Nicollet Mall, Minneapolis, MN 55403, and the copyright notice may not be removed without the consent of Retek Inc.

Information in this documentation is subject to change without notice.

Retek provides product documentation in a read-only-format to ensure content integrity. Retek Customer Support cannot support documentation that has been changed without Retek authorization.

The functionality described herein applies to this version, as reflected on the title page of this document, and to no other versions of software, including without limitation subsequent releases of the same software component. The functionality described herein will change from time to time with the release of new versions of software and Retek reserves the right to make such modifications at its absolute discretion.

Retek[®] Service Layer[™] is a trademark of Retek Inc.

Retek and the Retek logo are registered trademarks of Retek Inc.

This unpublished work is protected by confidentiality agreement, and by trade secret, copyright, and other laws. In the event of publication, the following notice shall apply:

©2004 Retek Inc. All rights reserved.

All other product names mentioned are trademarks or registered trademarks of their respective owners and should be treated as such.

Printed in the United States of America.

Customer Support

Customer Support hours

Customer Support is available 7x24x365 via email, phone, and Web access.

Depending on the Support option chosen by a particular client (Standard, Plus, or Premium), the times that certain services are delivered may be restricted. Severity 1 (Critical) issues are addressed on a 7x24 basis and receive continuous attention until resolved, for all clients on active maintenance. Retek customers on active maintenance agreements may contact a global Customer Support representative in accordance with contract terms in one of the following ways.

Contact Method	Contact Information
----------------	---------------------

E-mail	support@retек.com
--------	-------------------

Internet (ROCS)	rocs.retek.com Retek's secure client Web site to update and view issues
-----------------	---

Phone	+1 612 587 5800
-------	-----------------

Toll free alternatives are also available in various regions of the world:

Australia	+1 800 555 923 (AU-Telstra) or +1 800 000 562 (AU-Optus)
France	0800 90 91 66
Hong Kong	800 96 4262
Korea	00 308 13 1342
United Kingdom	0800 917 2863
United States	+1 800 61 RETEK or 800 617 3835

Mail	Retek Customer Support Retek on the Mall 950 Nicollet Mall Minneapolis, MN 55403
------	---

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.

Contents

Chapter 1 – Introduction	1
RSL Overview	1
RSL Javadoc	1
Chapter 2 – Technical Architecture	3
Overview	3
Architecture Layers	3
SAL and SPL Offered Interfaces and APIs	4
J2EE Application Model	5
Oracle PL/SQL-based Application model	6
Chapter 3 – Basic Operations	7
Simple service call	7
Chapter 4 – Common libraries	9
Platform	9
RSL/Integration	9
Third party provided	10
Chapter 5 – Configuration files	13
Services Framework Configuration	13
jndi_providers.xml	13
services_<app>.xml	13
service_flavors.xml	13
Other Configuration	13
commons-logging.properties	13
hibernate.cfg.xml	13
implfactory.properties	13
Log4j Configuration	14
log4j.dtd	14
log4j.xml	14

Chapter 6 – “Client” How to Guide	15
Client Application Layer Configuration	15
retек/services_<app>.xml.....	15
retек/service_flavors.xml	15
retек/jndi_providers.xml	15
Client Application Layer Code	16
Chapter 7 – “Provider” How to Guide.....	17
Service Provider Application Configuration	17
retек/services_<app>.xml.....	17
retек/service_flavors.xml	17
Service Provider Application Layer Code	17

Chapter 1 – Introduction

This manual is designed for System Administrators, Developers, and Applications Support personnel. Its purpose is to provide a basic understanding of the Retek Service Layer components, including the flow of a synchronous call between two applications. This chapter describes the components that make up the Retek Service Layer (RSL). These components can be distributed via an application server (IBM WebSphere or JBoss) or can be used in a stand alone environment.

RSL Overview

RSL handles the interface between a client application and a server application. The client application typically runs on a different computing 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 Retek 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.

RSL Javadoc

Javadoc is the tool from Sun Microsystems that generates API documentation in HTML format. The RSL implementators receive Javadoc documentation generated from Retek Service Layer code. Click the HTML file named 'index' in the applicable Javadoc zip file to open the Javadoc.

Chapter 2 – Technical Architecture

Overview

This chapter describes the overall architecture of the Retek 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 as the integration team provides them with wrapper classes that encapsulate the calls for retrieving a service proxy. The developer only needs to implement the code that creates an instance of the wrapper 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 code developed will be independent of service location – i.e. the exact same code will be able to run in a stand-alone java command line implementation or within a J2EE Stateless Session EJB environment.
- The Service Access Layer (SAL). This layer is generated by the Integration Group. 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. However, different implementations of the SAL will be needed by the CAL developer or Client application deployer, depending on the local or remote location of the required services.
- The Service Integration Layer (SIL). This layer provides a generic set of interfaces and utilities for accessing a service. These interfaces will provide logging of input and output parameters, timing log entries, client identification strings, etc. The full set of interfaces and capabilities is yet to be determined. The SIL is responsible for handling the intricacies of J2EE Applications finding and calling remote services or not.
- The Service Provider Layer (SPL). This layer is implemented by the developers belonging to the service provider or knowledgeable in the service application domain. 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.



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” Stateless Session Bean will be created and deployed that can handle all RMS provided simple services. However, it **must** be possible to deploy multiple copies of this bean so that different pools of different beans are responsible for different sets of services.

SAL and SPL Offered Interfaces and APIs

The SAL is responsible for offering a typed set of service interfaces to a Client Application (the CAL). The SPL is responsible for implementing a set of service interfaces. For J2EE deployed services, both layers will reference the same interface. For Oracle applications,

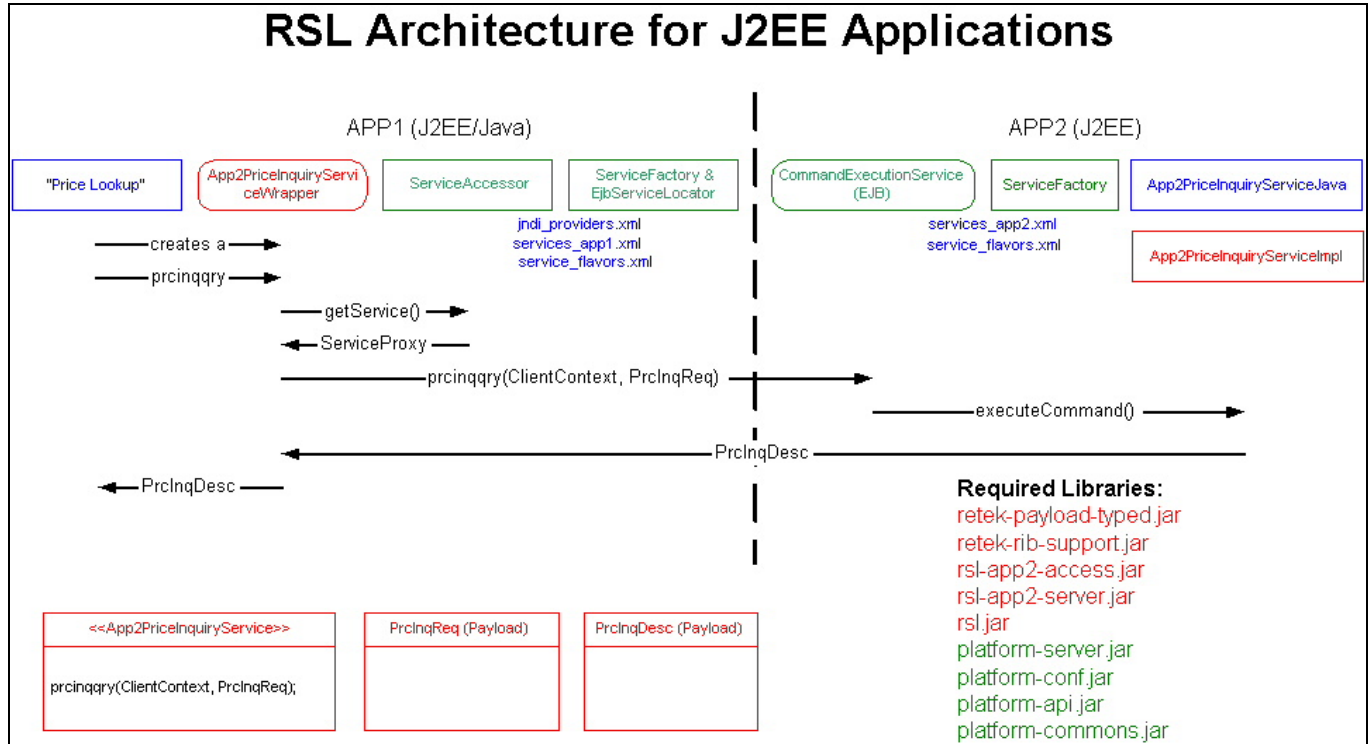
- Each separate interface will map to a PL/SQL package
- Each interface method will map to a specific PL/SQL stored procedure
- Each field in the input and output parameters will map to an Oracle Object fields in a one-to-one manner.



Note: The services offered by a single SSB (J2EE applications) or a single PL/SQL package (Oracle Forms based applications) 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.

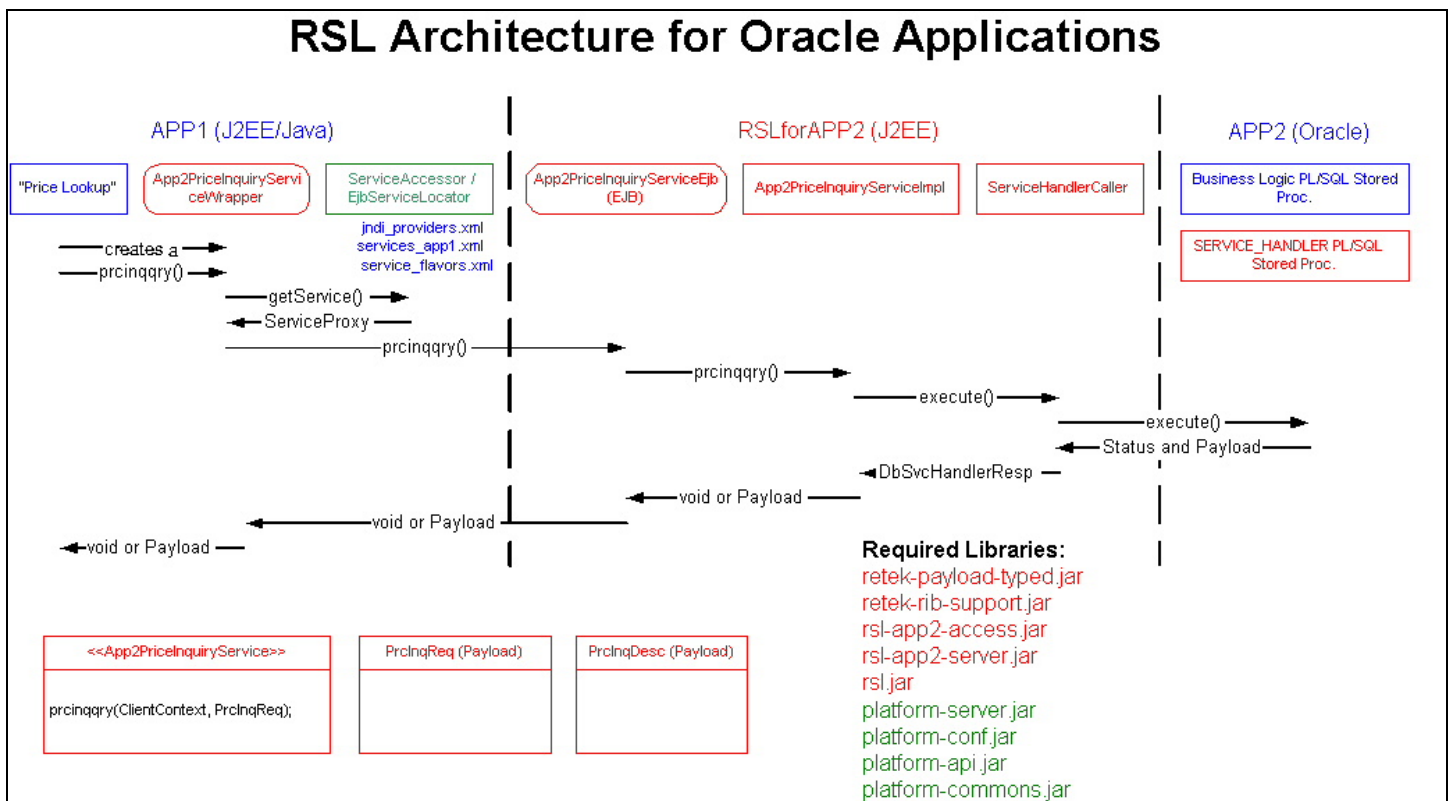
J2EE Application Model

The following diagram 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 integration team to both developers. Retek’s platform objects are denoted in green.



Oracle PL/SQL-based Application model

The following diagram shows the model and workflow to be used when integrating RSL with an application based on Oracle PL/SQL language (e.g. 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 “service provider” developer (right side.) The red objects indicate the interfaces, classes and payload objects provided by the integration team to both developers. Retek’s platform objects are denoted in green.



Chapter 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 internally 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 and calls the service.

At this point, the J2EE application server infrastructure takes control and performs a remote method invocation on the desired Stateless Session Bean. The method then performs the desired business logic and returns.

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. For the RSL, the service references a configuration file (hibernate.cfg.xml) to access the correct resource or database connection.

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.

Chapter 4 – 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 Retek. Its reuse is essential to interoperability between the RSL and other applications

Platform

platform-server.jar

This jar contains platform specific code for Retek. This includes the Service framework and various Hibernate related objects. All services developed will be dependant on this jar for its use of AbstractService and Service related classes.

platform-api.jar

This jar contains platform specific code for Retek. This includes Exceptions and other Objects that could be transferred “over the wire”.

platform-conf.jar

This jar contains platform specific configuration files.

platform-common.jar

This jar contains platform specific code for Retek.

RSL/Integration

retex-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.

retex-rib-support.jar

This jar contains various utilities for use with Retek 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) and Wrapper(s) to access Services from another application (SPL). These classes will ultimately be automatically generated by the RIB Framework Gui.

rsl-<service provider application>-server.jar

This jar contains specific RSL implementations of the Interface(s). The CommandExecutionServiceEjb will look up these Implementations. These classes essentially “wrap” the SPL’s implementation of the Interface(s).

rsl-<service provider application>-ejb.jar

There may be times when the Platform CommandExecutionServiceEjb is not available or applicable. In that case this EJB module jar will need to be included and deployed in the SPL’s application.

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 and within Hibernate. 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.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/>

hibernate2.jar

This jar contains hibernate related classes. Hibernate abstracts the database connection from the application. It also provides relational persistence, but is not used within the RSL. Documentation and related information is located at <http://www.hibernate.org/>

dom4j.jar

This jar contains XML DOM processing ability. It is needed for processing of the configuration file for hibernate. Documentation and related information is located at <http://www.dom4j.org/>

ehcache.jar

This jar contains caching related classes. It is needed by Hibernate for caching of objects in its relational persistence piece. Though the RSL does not use it, Hibernate will not function correctly without its presence.

commons-collections.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/>

cglib2.jar

This jar contains library classes for use by Hibernate. This includes transformation, reflection and bean utilities.

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 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>

Chapter 5 – Configuration files

The RSL uses a variety of configuration files. These are mostly platform 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 platform documentation for specific formatting of this file.

services_<app>.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 platform 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. On the SPL side, the RSL requires a flavorset of *businesslogic*. This is used to distinguish the correct implementation of business logic to use. Please reference platform documentation for specific formatting of this file

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

hibernate.cfg.xml

If required, this file must be in the classpath. It contains the hibernate configuration for database access. Hibernate is an abstraction tool to represent databases in a uniformed fashion. This is a platform standard. Here you can modify the database access to be via a container or JDBC. This file is only required for Oracle forms based service providing applications.

implfactory.properties

This file must be in the classpath. This is required for proper operation within a WebSphere environment. It includes the correct factory implementation for specific WebSphere handles.

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/>

Chapter 6 – “Client” How to Guide

Client Application Layer Configuration

As mentioned earlier, the CAL is developed by the “client application” developer. CAL development should take into account that it may be interfacing with multiple versions of the service provider. CAL implementations should insulate the rest of the client application from the service call specifics. The following files need to be properly configured in the client application.

retek/services_<app>.xml

An entry like the following should exist in the client application’s services_<app>.xml file. The exact package and app names will be as per the particular service being called.

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

retek/service_flavors.xml

At a minimum, the following entry must exist in the client application’s service_flavors.xml file.

```
<?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

At a minimum, the following entry must exist in the client application’s jndi_providers.xml file. The app name must match the app name from the services_<app>.xml file.

```
<?xml version="1.0" ?>
<ejb_context_overrides>
  <provider app="rpm" url="iiop://mspdev37:23813" factory="com.ibm.websphere.naming.
WsnInitialContextFactory">
  </provider>
</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, wrapper class and the input and output Payload classes will be provided to you by the Integration group.

One parameter seen below is the Retek Platform 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.service.ClientContext;
import com.retek.platform.util.type.security.SecureUser;
import com.retek.rib.binding.payload.PrcInqDesc;
import com.retek.rib.binding.payload.PrcInqReq;
import com.retek.rsl.rpm.PriceInquiryServiceWrapper;

// 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.
PriceInquiryServiceWrapper wrapper = new
PriceInquiryServiceWrapper(ctx);

// create the input payload object.
PrcInqReq request = createARequest();

// create the output payload object
PrcInqDesc price;

// call the query method on service
price = wrapper.prcinqqry(request);
```

Chapter 7 – “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.

retек/services_<app>.xml

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

retек/service_flavors.xml

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

Service Provider Application Layer Code

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 Interface+”Java” (see example above). This allows the Platform service framework to locate the SPL’s implementation class by using the “buisnesslogic” 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 PriceInquiryServiceJava() {
    super();
    // TODO Auto-generated constructor stub
}

public void attachFallbackHandler(FallbackHandler arg0) {
    LOG.debug("Executing business logic for
attachFallbackHandler");
    // TODO Auto-generated stub
}

public PrcInqDesc prcinqqry(ClientContext client, PrcInqReq
query) throws RetekBusinessException {
    LOG.debug("Executing business logic for query");
    // TODO Auto-generated method stub...add business logic here
    PrcInqDesc out = new PrcInqDesc();
    return out;
}
}
```