Oracle® Retail Integration Bus

Implementation Guide

Release 13.0.3

September 2009

ORACLE®

Oracle Retail Integration Bus Implementation Guide, Release 13.0.3

Primary Author: Susan McKibbon

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 software component known as **ACUMATE** developed and licensed by Lucent Technologies Inc. of Murray Hill, New Jersey, to Oracle and imbedded in the Oracle Retail Predictive Application Server - Enterprise Engine, Oracle Retail Category Management, Oracle Retail Item Planning, Oracle Retail Merchandise Financial Planning, Oracle Retail Advanced Inventory Planning, Oracle Retail Demand Forecasting, Oracle Retail Regular Price Optimization, Oracle Retail Size Profile Optimization, Oracle Retail Replenishment Optimization applications.

(ii) 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.

(iii) the **SeeBeyond** component developed and licensed by Sun MicroSystems, Inc. (Sun) of Santa Clara, California, to Oracle and imbedded in the Oracle Retail Integration Bus application.

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

(v) the software component known as **Crystal Enterprise Professional and/or Crystal Reports Professional** licensed by SAP and imbedded in Oracle Retail Store Inventory Management.

(vi) 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.

(vii) 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.

(viii) the software component known as **Style Report™** developed and licensed by InetSoft Technology Corp. of Piscataway, New Jersey, to Oracle and imbedded in the Oracle Retail Value Chain Collaboration application.

(ix) the software component known as **DataBeacon™** developed and licensed by Cognos Incorporated of Ottawa, Ontario, Canada, to Oracle and imbedded in the Oracle Retail Value Chain Collaboration 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

4 Oracle Retail Application APIs

5 Pre-Implementation Considerations

6 Deployment Architecture and Options

7 Implementation Process

Preface

The Oracle Retail Integration Bus Implementation Guide provides detailed information that is important when implementing RIB.

Audience

The Implementation Guide is intended for the Oracle Retail Integration Bus application integrators and implementation staff, as well as the retailer's IT personnel.

Related Documents

For more information, see the following documents in the Oracle Retail Integration Bus Release 13.0.3 documentation set:

- *Oracle Retail Integration Bus Installation Guide*
- *Oracle Retail Integration Bus Release Notes*
- *Oracle Retail Integration Bus Operations Guide*
- *Oracle Retail Integration Bus Integration Guide*

Customer Support

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

- https://metalink.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 recreate
- Exact error message received
- Screen shots of each step you take

Review Patch Documentation

If you are installing the application for the first time, you install either a base release (for example, 13.0) or a later patch release (for example, 13.0.2). If you are installing a software version other than the base release, be sure to read the documentation for each patch release (since the base release) before you begin installation. Patch documentation can contain critical information related to the base release and code changes that have been made since the base release.

Oracle Retail Documentation on the Oracle Technology Network

In addition to being packaged with each product release (on the base or patch level), all Oracle Retail documentation is available on the following Web site (with the exception of the Data Model which is only available with the release packaged code):

http://www.oracle.com/technology/documentation/oracle_retail.html

Documentation should be available on this Web site within a month after a product release. Note that documentation is always available with the packaged code on the release date.

Conventions

The following text conventions are used in this document:

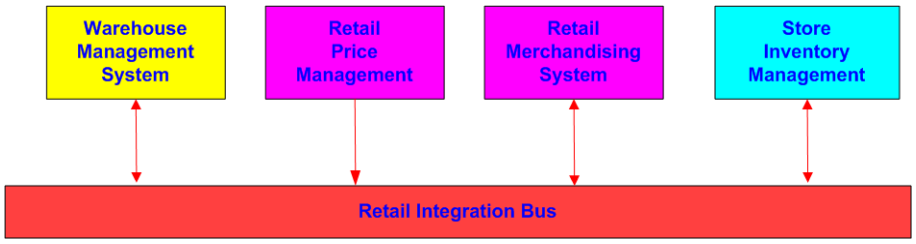
Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction

The Oracle Retail Integration Bus (RIB) is a fully distributed integration infrastructure that uses Message Oriented Middleware (MOM) to integrate applications. RIB enables various Oracle Retail applications to integrate in an asynchronous and near real time fashion. RIB provides additional value added business and infrastructure services to the Oracle Retail applications in addition to providing integration connectivity.

Each of the Oracle Retail Applications has their own implementation and deployment strategies and approaches, as well as individual integration touch points defined. The implementation of the RIB has to take into account the overall Oracle Retail application enterprise deployment architecture and try to fit into the model seamlessly.



The RIB acts as a shared communication layer for connecting various Oracle Retail applications and external applications throughout an enterprise computing infrastructure. It supplements the core asynchronous messaging backbone with additional application functionality such as intelligent transformation, routing and error handling.

Communication across the RIB is via xml messages (payloads). These payloads describe the retail business objects (such as items, purchase orders, suppliers, and so on) in a standard way and are governed by RIB on behalf of the Oracle Retail applications.

The RIB architecture is based on standard Java EE components and the Java Message Service (JMS). JMS is an integral part of the Java EE (Java Enterprise Edition) Technology stack. This is very different from the previous RIB releases that were based on a centralized model implemented using eGate proprietary components.

The integration solution provided by RIB system is made up of multiple Java EE RIB applications (rib-<app>.ear) that are autonomous in their execution behavior, and are deployed in a fully distributed topology. Even though they (rib-<app>.ear) are distributed and autonomous they communicate and coordinate messages with each other and works to provide the final asynchronous integration solution that the enterprise expects. The issues and considerations needed to properly deploy and configure it within an enterprise are the subject of this guide.

2

Standards and Specifications

This release of the RIB is designed and built on industry standard non-proprietary Java EE concepts and standards.

Java Platform Enterprise Edition (Java EE)

Java Platform Enterprise Edition (Java EE) is an umbrella standard for Java's enterprise computing facilities. It bundles together technologies for a complete enterprise-class server-side development and deployment platform in java.

Java EE specification includes several other API specifications, such as JDBC, RMI, Transaction, JMS, Web Services, XML, Persistence, mail, and others and defines how to coordinate among them. Java EE specification also features some specifications unique to enterprise computing. These include Enterprise JavaBeans (EJB), servlets, portlets, JavaServer Pages (JSP), Java Server Faces (JSF) and several web service technologies.

A Java EE "application server" manages transactions, security, scalability, concurrency, pooling, and management of the EJB/Web components that are deployed to it. This frees the developers to concentrate more on the business logic/problem of the components rather than spending time building scalable, robust infrastructure to run on.

Java EE Server

Oracle Application Server implements the Java EE specification and is the Java EE server vendor for RIB in this release. Oracle Application Server provides many additional services beyond the standard services required by the Java EE specification.

See Oracle® Application Server documentation for more information.

Java Message Service (JMS)

The Java Message Service (JMS) defines the standard for reliable Enterprise Messaging. Enterprise messaging, also referred to as Messaging Oriented Middleware (MOM), is universally recognized as an essential tool for building enterprise applications. By combining Java technology with enterprise messaging, the JMS API provides a powerful tool for solving enterprise computing problems.

See `http://java.sun.com/products/jms`.

Enterprise messaging provides a reliable, flexible service for the asynchronous exchange of critical business data and events throughout an enterprise. The JMS API adds to this a common API and provider framework that enables the development of portable, message based applications in the Java programming language.

The JMS API improves programmer productivity by defining a common set of messaging concepts and programming strategies that will be supported by all JMS technology-compliant messaging systems.

The JMS API is an integral part of the Java Enterprise Edition platform, and application developers can use messaging with components using Java EE APIs ("Java EE components").

JMS Provider

A JMS Provider is a vendor supplied implementation of the JMS interface, such as Oracle AQ JMS or OC4J JMS. Oracle Streams AQ implements the JMS specification and is the certified JMS provider for RIB in this release. AQ is built on top of the Oracle Database 10g Enterprise Edition.

See Oracle Database Enterprise Edition documentation for AQ information.

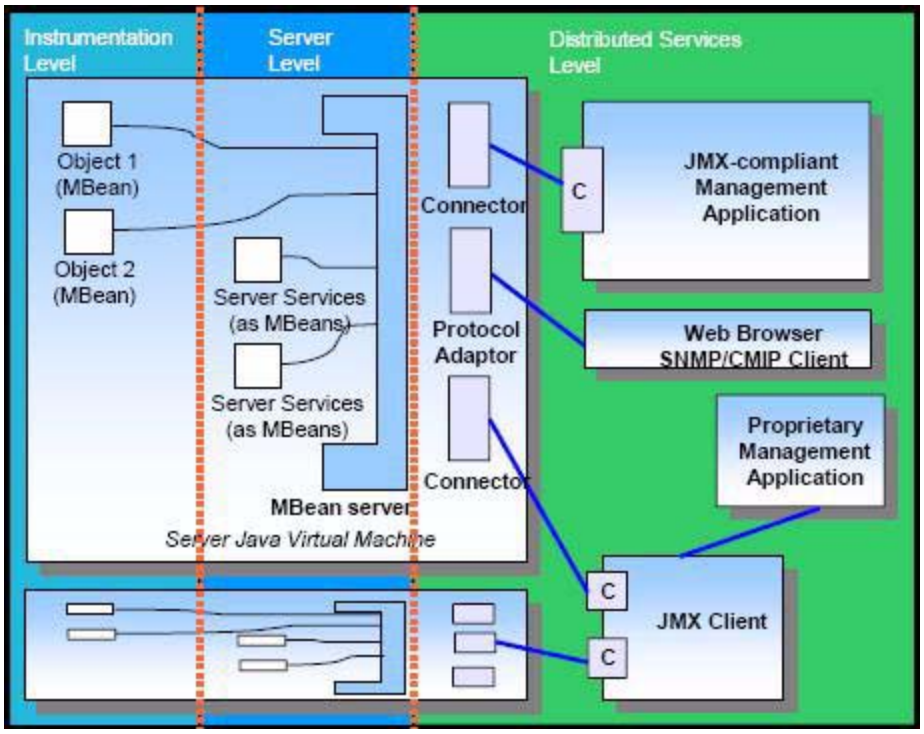
Java Management Extensions (JMX)

The RIB is a backend, headless application that does not need active business user participation for its daily operations. When the environment is stable there is no user intervention required for the system to keep running. For such a backend system it is critical that there are proper alerting and notification mechanisms built into the application for situations when the system runs into trouble or to communicate interesting business situations to administrators.

Java Management Extensions (JMX) is a specification to provide management and monitoring capabilities to applications that are built using java programming language.

The JMX is based on a three-level architecture:

- The Probe/Instrumentation level: This layer contains the probes (called MBeans) that instrument the application resources and make the resource available through an agent layer.
- The Agent level: The MBeanServer is at the core of JMX infrastructure. It is a registry/catalog of all MBeans available for management.
- The Remote Management level: This layer enables remote applications to access the MBeanServer through Connectors and Adaptors. A connector provides full remote access to the MBeanServer API using various RPC communication protocol like RMI, IIOP, WS-*, and others. A JMX adapter on the other hand adapts the JMX API and events to other standard protocol like SNMP or provide a web based GUI (HTML/HTTP) interface to the JMX API/Events.



In addition to the three layers presented in the architecture, JMX provides a notification model that follows the observer observable design pattern. By using notifications, JMX agents and MBeans can send alerts or report information to third party management applications. Users can receive notifications as a way of being informed of critical events or requests for attention.

Since efficient management and monitoring of RIB components are essential to the RIB product, and also seamless integration to standard third party enterprise management tools was a requirement, the RIB application has been fully instrumented to be manageable by any JMX compatible management tools.

The RIB adapters can be controllable using standard JMX tools like OAS Enterprise Manager and JConsole. When interesting business activity happens inside RIB, the RIB components emit alerting events to the RIB alerting framework. By default the alerting framework is configured to send JMX and Email alert notifications. Anyone interested in RIB's JMX alerts can subscribe to RIB notification types using their choice of JMX compatible management tools. JMX management tools provide a way to configure your listener/handler in the tool to react to the incoming alert event.

Note: See JMX management tool vendor documentation for how to add listeners to JMX alerts.

3

Core Concepts

The RIB is designed as an asynchronous publication and subscription messaging integration architecture. This allows the decoupling of applications and their systems. For example, a publishing application need not know about the subscribing applications, other than the requirement that at least one durable subscriber must exist. It decouples the systems operationally. Once a subscriber is registered, the RIB persists all published messages until all subscribers have seen them.

The publishing adapter does not know, or care, how many subscribers are waiting for the message, what types of adapters the subscribers are, what the subscribers' current states are (running or stopped), or where the subscribers are located. Delivering the message to all subscribing adapters is the responsibility of the RIB with the help of the underlying JMS server.

Physically, the message must reside somewhere so that it is available until all subscribers have processed it. The RIB uses the JMS specification for its messaging infrastructure. The JMS accepts the message from the publisher and saves it to stable storage, a JMS topic, until it is ready to be picked up by a subscriber. In all cases, message information must be kept on the JMS until all subscribers have read and processed it.

The RIB interfaces are organized by message family. Each message family contains information specific to a related set of operations on a business entity or related business entities. The publisher is responsible for publishing messages in response to actions performed on these business entities in the same sequence as they occur.

Each message family has specific message payloads based on agreed upon business elements between the Oracle Retail applications.

Key Functional Requirements

The design and architecture of the RIB infrastructure is based on two key requirements driven by the Oracle Retail application business model.

Guaranteed Once-and-Only-Once Successful Delivery

The RIB must preserve and persist all business events (messages) until all applications (subscribers) have looked at the message and have successfully consumed it or decided they do not care about that event (message). In other words RIB must deliver to every subscriber all messages except those filtered per a subscribing application's requirements.

A business event (message) must be redelivered to the consumer application if the business event (message) was not consumed successfully. The redelivery process is bound by the same rules of sequencing as normal (non-redelivered) business event (message).

Preservation of Publication Sequence

The business event (message) must be delivered to all the subscribing applications in the order (FIFO) the business event (messages) was published by the publishing application.

To enable this, the publishing application defines a business object ID whose existence informs RIB that this and all subsequent messages with the same business object ID have to be processed in order. Business event (message) ordering (FIFO) is assured only for messages with the same business object ID within the same message family.

Message Family and Message Types

The RIB messaging adapters and payloads are designed around the concept of a message family.

Each RIB message belongs to a specific message family. Each message family contains information specific to a related set of operations on a business entity or related business entities. The publisher is responsible for publishing messages in response to actions performed on these entities in the same sequence as they occur.

One example of a message family is the orders message family used to contain information about purchase order events.

A message family may contain multiple message types. Each message type encapsulates the information specific to a business entity within one or more business events. For example, the order message family is published for events such as "Create PO Header", "Create PO Detail", "Update PO Header", or "Delete PO Detail"

A single business event, such as updating a purchase order, may involve multiple business entities, such as a line item within the purchase order.

Because a single business event may involve multiple business entities, the application may publish messages for this event from multiple message families for a single business transaction. More than one message type within a message family may also be created.

There are two broadly defined types of functional interfaces in the RIB (message families); foundation data and transactional data.

Foundation Messages

After populating application tables with initial company seed data, item foundation information is needed. Foundation messages are defined as those with payload that carry basic product data.

This table is an example from the RIB Integration Guide.

Functional Area	Publishing Applications	Subscribing Applications
Items	RMS	RWMS, SIM
Item Locations	RMS	SIM
Locations	RIB	RWMS
Stores	RMS	RWMS, SIM
Vendor	RMS	RWMS, SIM
Warehouses	RMS	RWMS, SIM

Transactional Messages

After populating application tables with initial seed data and after all required item foundation data messages have been subscribed to, all applications are prepared to publish and subscribe transaction data messages. Transactional messages communicate business events involving two or more organizations within a retail supply chain, for instance, between Oracle Retail Merchandising System (RMS), Oracle Retail Store Inventory Management (SIM), and Oracle Retail Warehouse Management System (RWMS), external suppliers and financial systems.

This table is an example from the RIB Integration Guide.

Functional Area	Publishing Applications	Subscribing Applications
Allocations	RMS	RWMS
Appointments	RWMS	RMS
ASN Outbound	RWMS, SIM	RMS, SIM, RWMS
ASN Inbound	RWMS, External	RMS, SIM, RWMS
Inventory Adjustments	RWMS, SIM	RMS
Inventory Request	SIM	RMS
Receipts	RWMS, SIM	RMS
Purchase Order	RMS, SIM	RWMS, SIM
Stock Order Status	RWMS, SIM	RMS
Transfers	RMS	RWMS, SIM

RIB Message Envelope and Payloads

Whenever a publishing application adapter publishes a message, it wraps the message in an envelope known as the RIB message envelope. The envelope is a standard message delivery format where the message information, the data payload, is contained within the overall delivery information. The envelope itself provides information that the RIB uses, such as RIB hospital information and routing information.

Message Life Cycle

The publishing application is responsible for creating the initial message contents. The RIB publishing adapter publishes it to the JMS Server and makes it available to any JMS subscribers. The RIB knows what subscribers are to receive the message due to the RIB configuration—this configuration associates a set of subscribers to each publisher and message family combination.

For PL/SQL Applications, database tables associated with the publishing application typically stage message information. One or more RIB publishing adapters poll the application via a stored procedure call. For Java EE Applications, the application calls a RIB Enterprise Java Bean (EJB) with the payload information to be published.

The message resides on a Java Message Service (JMS) immediately after publication. The JMS topic provides stable storage for the message in case a system crash occurs before all message subscribers receive and process it.

A fundamental RIB system requirement is that a message must be delivered to and processed successfully exactly once by each subscriber. Furthermore, all work performed by the subscriber and the RIB must be atomically committed or rolled back, even if the JMS server is on a remote host. The standard way to perform this is by using an XA compliant interface and two-phase commit protocol.

After initial publication, a message may undergo a series of transformation, filtering, or routing operations. A RIB component that implements these operations is known as a Transformation and Address Filter/Router (TAFR) component. TAFR is the acronym for Transform, Address, Filter, and Route. A TAFR is completely internal to the RIB and does not reside in either the publishing or subscribing application. The RIB performs these intermediate transformation and routing operations on some messages before making them available to the subscribing application.

A single TAFR may only transform a given message, only filter the message, only route it, or combine any of the three operations.

- Transform - A message may be transformed from one message type into another, for example, 'WH' (warehouse) from RMS to 'Location' for RWMS.
- Filter - A message may be filtered. Filtering can occur based on message type or based on content.
- Route - A TAFR may route a message. For example, whenever a stock order message is published for a warehouse with an instance of RWMS, the TAFR routes it to the particular RWMS instance from where the stock will be fulfilled and not to warehouses that do not stock the order's items.

TAFR operations are specific to the set of subscribers to a specific message family. Multiple TAFRs may process a single message for a specific subscriber and different specific TAFRs may be present for different subscribers. Different sets of TAFRs are necessary for different message families. If all subscribers to a message can process all messages within a message family without any TAFR operations, then no TAFR components are needed.

Message processing continues until a subscribing adapter successfully processes the message or determines that no subscriber needs this message.

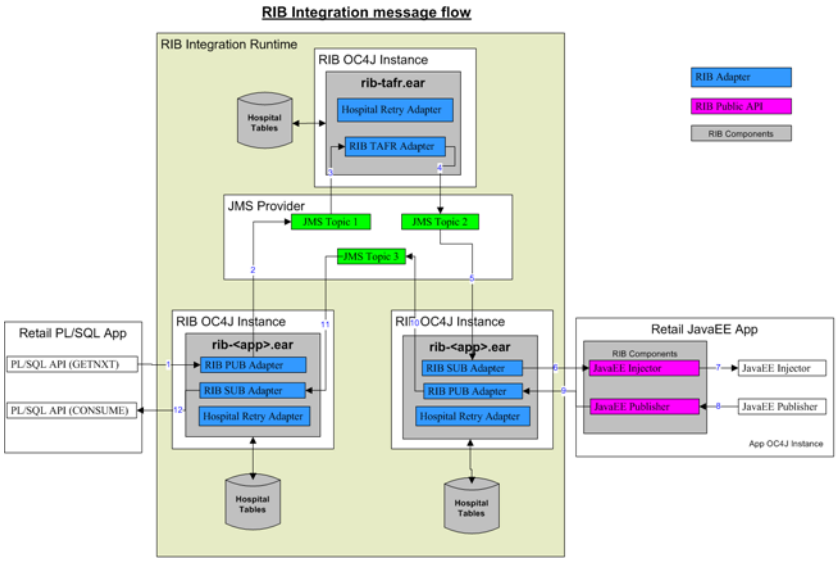
When a subscriber gets a message to be processed, the adapter checks to see if the RIB Hospital contains any messages associated with the same entity as the current message. If so, then the adapter places the current message in the hospital as well. This is to ensure messages are always processed in the proper sequence. If proper sequencing is not maintained, then the subscribing application's data can get corrupt.

If an error occurs during message processing, the subscribing adapter notes this internally and rolls back all database work associated with the message. When the message is re-processed (since it has yet to be processed successfully), the adapter now recognizes this message is problematic and checks it into the hospital.

After a message is checked into the RIB Hospital, a retry adapter extracts the message from the hospital and re-publishes it to the JMS topic for reprocessing. The message remains in the hospital during all re-tries until the subscribing adapter successfully processes it.

Messaging Components

The RIB is a messaging system made-up of components that are packaged and shipped as an integration solution between the Oracle Retail applications. The application boundary between RIB and Oracle Retail applications can be confusing at times, so this section defines the RIB components and their responsibility and ownership. A diagram illustrating the RIB integration message flow follows:



RIB Subsystem Components

This section describes the components of the RIB subsystem.

Adapters

A RIB adapter is a component that coordinates business event (message) generation and processing with the respective Oracle Retail application interface. Each adapter in the RIB is created to handle a specific functional interface. RIB adapters are developed using Enterprise Java Beans (EJB) components architecture, subscribing adapters use Message Driven Beans (MDBs) and publishing adapters use Stateless Session Beans (SLSBs).

The RIB provides four types of adapters that Oracle Retail applications can exploit to integrate with one another. These adapter types are: publisher, subscriber, TAFR, and hospital retry. They have been built using different technologies based on their particular needs.

Subscriber and TAFR adapters use Message Driven Bean (MDB) technology to register with JMS topics and receive messages for further processing.

Publisher and hospital retry adapters make use of the Java SE (Standard Edition) timer facility to schedule repetitive events that trigger calls to Stateless Session Beans (SLSBs) to query application tables for messages to publish to the JMS server.

As stated in the introduction, a fifth type of adapter exists for publishing messages in a pushing fashion. The Oracle Retail applications invoke this adapter at will for publishing messages.

These adapters have not been considered part of the scope of this technical document in regard to providing a mechanism for starting and stopping them.

Due to the variety of technologies used by the adapters, the goal of this technical design has been to isolate users from these differences and provide them with a common management interface that can be used to control the state of the adapters. During the last few years, the Java Management Extensions (JMX) specification has become a well known standard that defines the management layer for enterprise Java applications. JMX defines standard methodologies for declaring enterprise application components as manageable resources that can be exposed in a consistent way such that any JMX compliant management application can access and provide means for control.

JMS Domains, Destinations, Subscriptions

JMS defines two types of messaging domains: point-to-point and publish/subscribe. RIB uses publish/subscribe types of messaging domains for all its communication. Publish/subscribe is a one-to-many type of message distribution model where one source application en-queues the message and many destination applications can de-queue the same message and process independently of the other peer applications. In publish/subscribe the destinations are known as topics, the en-queuer application is known as publisher, and the de-queuer is known as subscriber. Unlike point-to-point, in publish/subscribe the publisher and subscriber are totally ignorant of each other and do not and should not know about each others existence. The JMS Topics retain the messages only as long as it takes to distribute them to current active (running) subscribers. There is also a timing dependency between publishers and subscribers. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages. The JMS specification relaxes this timing dependency to some extent by allowing clients to create durable subscriptions. By creating durable subscriptions the JMS server will continue to hold the messages for all registered subscribers for that topic until the subscriber consumes the message or deletes the subscription. There are two types of subscribers, non-durable and durable subscribers. The RIB uses only durable subscribers which allows the Oracle Retail edge applications to be in up or down state independently but still not loose any messages and catch up when the application comes back up. Every subscribing RIB adapter registers its durable subscriber with a subscription name that contains its rib-<app> application name and the adapter name in it.

RIB defines logical grouping of retail specific business objects (BO) and business functions in a concept called message family. For every message family there is a corresponding JMS topic. These JMS topics are used as communication pipelines between the source and destination Oracle Retail applications for exchanging the business objects.

The list of JMS topics used by RIB components is detailed in the RIB Integration Guide - Reports.

JMS Message Selector

A key aspect of the JMS usage that the RIB relies on is the attachment of message properties to published messages and the use of selectors by message subscribers. Message properties are used to convey information about the message outside of the actual message data to establish a logical channel for messages.

JMS message selectors are used by the RIB to filter the messages that each subscriber picks up. In other words, using the message properties, selectors act as a filter to weed out messages a subscriber should not process.

The message property set and used by the RIB messages is called threadValue. The thread value is associated with a logical channel of a message stream. All messages for a specific family with a specific business object ID always contain the same threadValue property. This, combined with the standard first in, first out (FIFO) message ordering on the topic, is integral to message sequencing. Messages with different threadValue properties are not guaranteed to be processed in the same relative order as publishing.

Messages published without JMS Message Property present will not be picked up by the standard subscribing RIB adapters.

Additional RIB JMS Message Properties

Every message published by the rib-<app> applications includes a number of JMS user defined header properties. In the current release, these properties are only set, not used by any RIB components. In the future, these properties will be used for intelligent performance enhancement and optimization and for traceability and auditability of RIB messages.

The message properties are as follows:

- Property Name: appName

  Type: java.lang.String

  Required Property: false

  Example: appName=rib-rms

  Description: The appName property contains the rib-<app> application name that published this particular message.

- Property Name: adapterInstance

  Type: java.lang.String

  Required Property: false

  Example: adapterInstance=Item_pub_1

  Description: The adapterInstance property contains the rib-<app> adapter instance name that published this particular message.

- Property Name: family

  Type: java.lang.String

  Required Property: false

  Example: family='Item'

  Description: The family property contains the RIB family name that this particular message belongs to.

- Property Name: needMessageOrderPreservation

  Type: boolean

  Required Property: false

  Example: needMessageOrderPreservation=true

  Description: This property will have a value of true if any ribMessage node within the RibMessages xml has a message that have businessObjectId set. This property will allow us to take advantage of the fact that now we know which messages need message order preserving at JMS header level(without opening the message). In the future, we will be able to take advantage of that information and further parallelize our processing and get better throughput without losing message sequencing.

- Property Name: topic

  Type: java.lang.String

  Required Property: false

  Example: topic='etItem'

  Description: This topic property contains the RIB topic name that this particular message is published to or subscribed from.

- Property Name: ribKernelVersion

  Type: java.lang.String

  Required Property: false

  Example: ribKernelVersion=13.0.3

  Description: The system determines the rib kernel jar version number at runtime and includes its value in this JMS property.

- Property Name: ribFuncArtifactVersion

  Type: java.lang.String

  Required Property: false

  Example: ribFuncArtifactVersion=13.0.3

  Description: This is a place holder for future enhancement. The idea is the system will somehow determine the runtime payload version and include that information in the message for better compatibility management. This property will be enhanced in a future release.

- Property Name: ribMessageCount

  Type: int

  Required Property: false

  Example: ribMessageCount=12

  Description: This property contains the number of ribMessage nodes there are in a RibMessages xml message. This value gives us some indication of message aggregation in play. It might be used in the future to better optimize message flow paths based on the size/number of the messages.

- Property Name: uuid

  Type: java.lang.String

  Required Property: false

  Example: uuid='116cfabd-8949-4f93-bb61-aaa88e168f30'

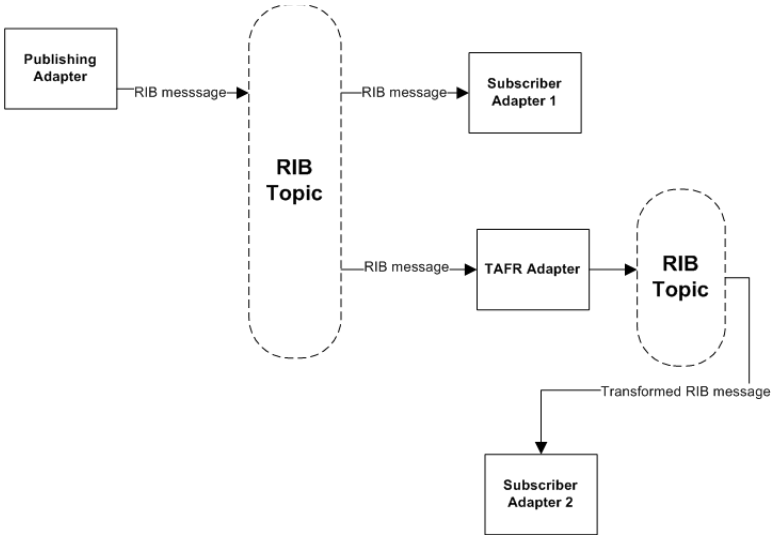
  Description: This property contains a universally unique identifier for every message. This unique identifier will provide better traceability of a message within the JMS system. This property complements the ribMessageID xml element that is there to trace messages within the RIB logs.

Simple Message Flow

The typical lifecycle of a message through the RIB is as follows:

1. First, the publishing adapter creates the message. The event that triggers the message creation may be a polling operation in case of PL/SQL applications or a synchronous invoke in case of Java EE applications. The message is published to a predetermined JMS topic.
2. The message is now available for all registered subscribers to the JMS topic for pick up. Subscription is based on the message family.
3. Once a subscriber gets the message, it is free to process it according to its own rules. In the case of a transformer adapter, the adapter can open the message, modify its contents, and then publish the modified message to a new topic. The source topic and destination topic that a TAFR uses must always be distinct/different topic. There may be new subscribers to the modified message, and the scenario repeated for each of these subscribers.
4. When each subscriber has finished (commit) processing a message, the JMS server updates the state of the message to reflect that it has been processed by this subscriber.
5. The JMS Server deletes the messages on the topic after delivering it to all the registered subscribers.

The figure below is a generalized view of a simple RIB message flow that involves all of the basic components. Two applications require this data and subscribe to it. One subscribing application requires a certain transformation be applied to the data, but the other subscriber can process the message without any transformations.



The RIB Hospital

The RIB Hospital is a collective term for a set of Java Classes and database tables whose purpose is to provide a mechanism to handle system and business related errors while meeting the fundamental RIB requirements:

- Guaranteed once-and-only-once successful delivery.
- Preservation of publication sequence (even in case of failures).

When a message is processed, the adapter checks to see if the RIB Hospital contains any messages associated with the same businessObjectId as the current message. If so, then the adapter places the current message in the hospital as well. This is to ensure messages are always processed in the proper sequence. If proper sequencing is not maintained, then the subscribing application's data can get corrupted.

If an error occurs during message processing, the subscribing adapter notes this internally and rolls back all work associated with the message. When the message is re-processed (since it has yet to be processed successfully), the adapter now recognizes this message is problematic and checks it into the hospital.

For Publication, there are some RMS publishers that return an 'H' status to denote a problem creating a new message for a specific business object. This status may be due to database locks being held by on-line users of an Oracle Forms application or it could also be due to some data incompatibility found in the GETNXT() procedure. Whenever a publisher recognizes that a message for a business object cannot be published due to one of these conditions, the message must go into the RIB Hospital.

After a message is checked into the RIB Hospital, a retry adapter extracts the message from the hospital and tries to re-publishes it to the integration bus.

RIB Hospital Dependency Check

The RIB Hospital dependency check logic assumes that each message family has a single unique businessObjectId for all business object entities its messages are associated with. This businessObjectId must be the same for the same business entity across all message types within the message family. If any message for a specific business entity is placed into the RIB Hospital, then the RIB Hospital dependency check logic automatically inserts any subsequent messages for the same business object. This is to preserve the message sequencing and guaranteed exactly once successful message processing. Otherwise, multiple update messages for a business object may be processed in an incorrect order and create incompatibilities between applications.

If the businessObjectid is not set, then there is no dependency check. Not all message families set the businessObjectId or it is not set on all message types. See Oracle Retail application documentation (for example, "Message Publication and Subscription Designs" in the *Oracle Retail Merchandising System Operations Guide, Volume 2*).

RIB Hospital Insert

If a message is to be inserted into the RIB Hospital because of an error during processing, it is sent to the subscribing adapter twice. This is because subscribing adapters are executed within the context of a distributed transaction, using the XA two-phase commit protocol. This transaction is controlled by the Java EE Application Server. If the RIB adapter returns success, the application server removes the message and all database work is committed. If the adapter returns failure, the message never leaves the integration bus topic and the database work is rolled back.

When the initial failure occurs while processing the message, the error is flagged within the RIB Hospital software, the adapter returns failure so that the database transaction is rolled back, and the message is kept on the integration bus topic.

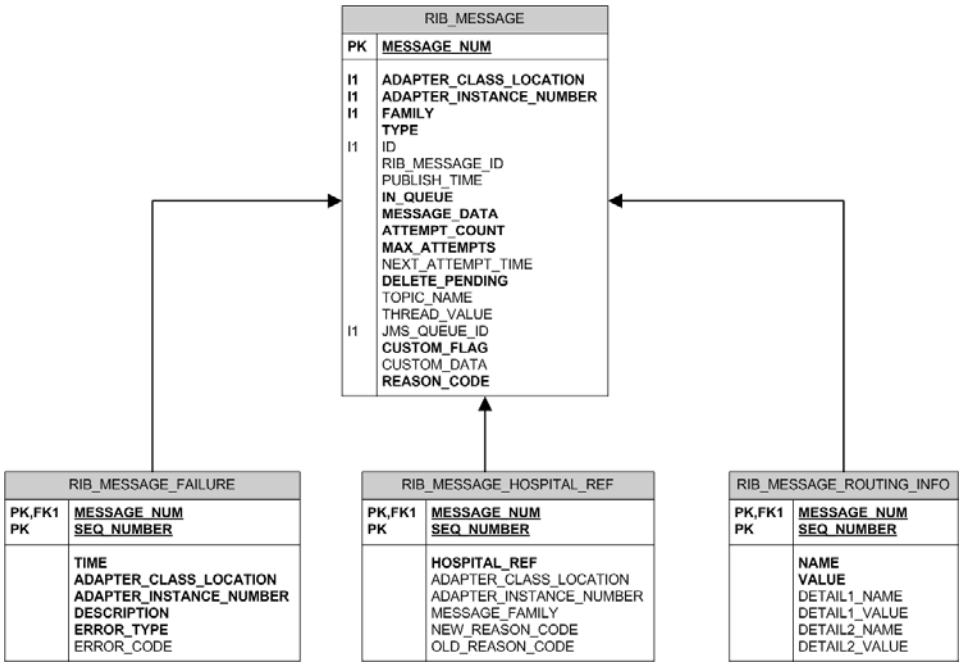
Note: The XA interface is a standard protocol between a transaction manager and a database or resource manager. Note that both the JMS topic connection and the database connection must support the XA protocol. For more information regarding the XA standard, see the URL, `http://www.opengroup.org`.

RIB Hospital Tables

The RIB Hospital tables are:

- RIB_MESSAGE - contains the message payload, all single-field envelope information, and a concatenated string made from <id> tags. It also contains a unique hospital ID identifying this record within the hospital.
- RIB_MESSAGE_FAILURE - contains all failure information for each time the message was processed.
- RIB_MESSAGE_ROUTING_INFO - contains all of the routing element information found in the message envelope.
- RIB_MESSAGE_HOSPITAL_REF - contains all of the hospital reference information found in the message envelope.

A database sequence, RIB_MESSAGE_SEQ, is used to maintain a unique message number associated with each message placed into the RIB Hospital.



These tables will have been created during the database portion of the Oracle Retail application install (for example, RWMS, SIM, RPM, AIP, or RMS).

The RIB Hospital tables are internal system tables that maintain the RIB runtime state of the system. The entries in these tables must not be manipulated by non RIB tools when the RIB is running.

RIB Hospital Retry

After a message is inserted into the RIB Hospital, the hospital retry adapter is used to re-post the message to the JMS in order to retry its processing. The assumption is that the error is a transitory one - records locked or there is an external dependency that has not been met. The number of times a message is retried is configurable.

The hospital retry is responsible for maintaining state information for hospital records - what has happened to the record or message information. Each time the message is re-processed, a record is kept of the event along with the results. The design is to provide a means to halt processing for messages that cause errors while allowing continued processing for the good messages.

One element of this information is whether the message has been queued to the JMS topic for re-try processing. Thus, manually deleting messages from the hospital database using SQL directly may produce severe processing problems. Similarly, deleting messages directly from the JMS provider may result in a message that is never retried again, as the logic in the retry assumes the message is queued within the JMS.

There are three kinds of hospital retry adapters:

- Sub Retry Adapter
- JMS Retry Adapter
- Pub Retry Adapter

All subscriber side retrying of messages are handled by the Sub Retry Adapter. The Sub Retry Adapter looks at all messages with reason code "SUB", then filters and identifies the messages that are ready to be reprocessed, keeping message ordering in mind.

Oracle Retail applications are not aware of the fact that the integrations of the business data is happening through a JMS server. RIB abstracts that fact that it is using a JMS server from the retail applications. When the JMS server is down or RIB has some problem publishing to the JMS server, RIB will not rollback the transaction as long as it is a recoverable problem. In such situation all messages are inserted to the RIB Hospital with a reason code of "JMS" and publications continues on. The JMS Retry Adapter retries all messages with reason code of "JMS" at a later time.

All messages with reason code of "PUB" are retried by the Pub Retry Adapter. RMS is the only retail application that needs the Pub Retry Adapter.

Hospital Attempt (Retry) Count

When the message first comes through the subscriber, if there is no businessObjectid, then there is no dependency check performed. If the message cannot be processed, it is then inserted into the hospital with an attempt_count = 1.

A message that comes through the subscriber that has a businessObjectid has a dependency check performed. If there is no dependency and the message cannot be processed, it Is then inserted into the hospital with an attempt_count = 1.

A message that comes through the subscriber that does match the ID and family of another message in the hospital is known to be dependent, so it goes to the hospital immediately, with an attempt_count = 0.

4

Oracle Retail Application APIs

PL/SQL Stored Procedure APIs

Each PL/SQL based application uses a Message Family Manager (MFM) specific API for publishing all messages within a specific message family. This API is the interface to a stored procedure package and wrappers the staging table and additional business logic surrounding the message publication.

The RIB Publishing Adapter polls the API by calling a routine in the MFM called "GETNXT()". The MFM "GETNXT()" PL/SQL stored procedure may contain simple or complex logic that is specific to the message types published. For example, a simple Create Vendor message may involve merely selecting and then deleting a single record from the vendor staging table. On the other hand, a Create Purchase Order message requires fairly complex logic to create because of the business process dependencies. Many changes may be made to a PO before it is approved.

The RIB Subscribing Adapter invokes the API by calling a routine in the MFM called CONSUME().

The RIB Pub Hospital Retry Adapter invokes the API by calling a routine in the MFM called PUB_RETRY().

Oracle CLOB APIs

The main facet of this API involves the use of Oracle CLOBs (Character Large Object Binaries) as the means to pass information to and from an Oracle stored procedure. The stored procedure is responsible for parsing or building the message payload.

There are only a few of this type remaining in RMS.

List of Interfaces.

API's using CLOBs have internal triggers that are fired when a specific database table is modified. The trigger retrieves all of pertinent information to create a specific type of message (XML payload) and inserts it into a staging table using an application specific Message Family Manager (MFM) API. The payload is contained in an Oracle Character Large Object Binary (CLOB). The staging table that holds the payload data must also maintain the following:

- The order that messages are created
- The CLOB containing the "payload" XML

- Any routing or filtering key values
- The message type associated with the business event that created the message. The message type is specific to the message family and a single business event may produce multiple messages of differing types within different families.

RIB_XML and RIB_SXW Database Packages

These PL/SQL packages contain utilities to make the generation and parsing of XML documents easier. It is based on Oracle's XDK, and is designed to support CLOB application-specific APIs that read and write XML messages.

Oracle Object API's

These application interfaces use Oracle Objects to pass information to and from the stored procedure. Each RIB Object corresponds to the XSD that defines the RIB Message payloads for that message family. This is the predominant type of PL/SQL API used in Oracle Retail integration via the RIB.

When a message is ready for publication, the Message Family Manager GETNXT() Stored Procedure examines its staging tables and creates the appropriate RIB Object for publication. In many cases, these staging tables contain columns that are themselves declared a specific type of RIB Object. Once the complete RIB Object is ready, the GETNXT() Stored Procedure returns this(RIB Object) to the calling RIB Adapter, which then converts the RIB Object into an equivalent XML string.

When a subscribing adapter gets a message from the JMS topic it constructs the Oracle Object by parsing the incoming payload xml. The newly created Oracle Object is passed in to the CONSUME() stored procedure to process the message.

RIB Related Database Tables

PL/SQL stored procedures use three tables to refine their behavior: RIB_SETTINGS and RIB_TYPE_SETTINGS and RIB_OPTIONS. Not all applications use these.

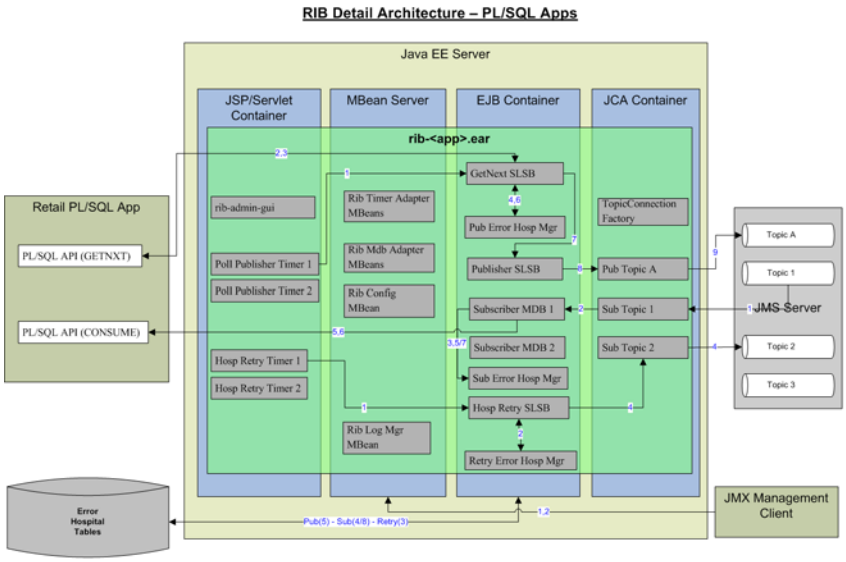
The RIB_SETTINGS table defines, on a per message family basis:

- The number of channels to use when publishing. See "Multi-channel" in the *Oracle Retail Integration Bus Operations Guide*.
- The maximum number of details to publish within a create, update, or delete message. Oracle Retail applications typically do not have a limit to the number of details a specific business object can have. Hence, a purchase order may be created containing tens of thousands of detail lines - each line a specific item/location combination. A single PO Create message containing 30,000 or so lines require a vast amount of resident memory to parse. This column limits the PO Create and subsequent PO Detail Add messages to a set number of details.

The RIB_TYPE_SETTINGS table is used internally by the application.

The RIB_OPTIONS table is used by the CLOB APIs for the creation of XML.

Detail Architecture - PL/SQL Apps

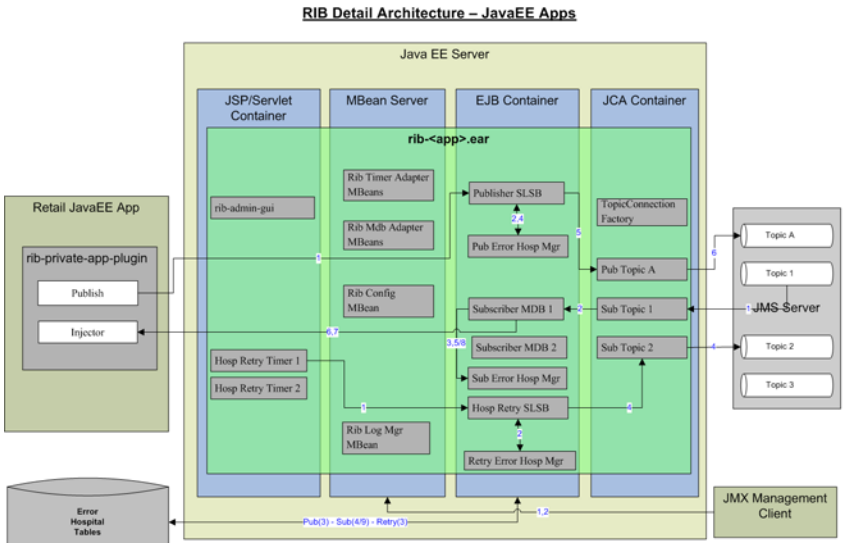


Oracle Retail Java EE APIs

These interfaces to the RIB are via Message Driven Bean (MDB) for subscribers and by Stateless Session Bean (SLSB) to publish messages to the JMS. This architecture uses Payload Java Beans to communicate event information from the RIB code to the application messaging processing logic.

The internal architecture of RIB is very similar between Oracle Retail PL/SQL applications and Oracle Retail Java EE applications. The only significant difference is in the publishing adapter types. For PL/SQL Retail applications RIB keeps on polling the stored procedure every few seconds to find out if there is any work. When the stored procedure returns some data (that is, when there is some work), RIB goes and does the work. In Oracle Retail Java EE applications RIB does not do any polling. The roles are reversed where the Oracle Retail application requests RIB to publish a message. Thus, there are two types of publishing adapters in RIB depending on the connecting Oracle Retail application type. The Java EE application uses request-driven publishers and the PL/SQL application uses timer-driven publishers.

Detail Architecture JavaEE Apps



API Return Status Codes

PL/SQL GETNEXT Return Codes

S - Success

N - No message

H - Hospital

E - Error

PUB_RETRY Return Codes

S - Success

N - No message

H - Hospital

E - Error

I - Keep calling

CONSUME Return Code

S - Success

E - Unhandled Error

5

Pre-Implementation Considerations

Before the RIB is installed into an enterprise, there are many factors that need to be considered. Planning and addressing each of the factors will avoid having to re-install or re-architect because of performance or operational problems.

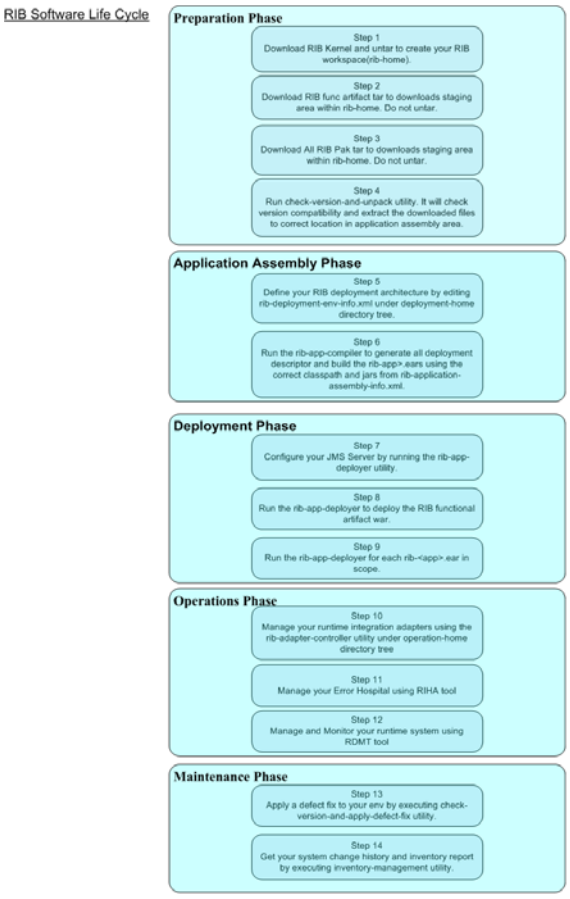
The process of RIB implementation requires the creation or modification of a retailer's Enterprise Integration Architecture. Typically, retailers will already have an integration strategy, plan or architecture and products in place to integrate their current systems.

The deployment of the RIB is always a portion of the deployment of the Oracle Retail applications, almost always with RMS. Because the implementation of RMS is a long cycle project, and always involves data conversions and integration into a retailer's existing infrastructure, the RIB implementation planning is strategic to that effort.

RIB Software Lifecycle Management

Software applications, after being made generally available (GA), have a well defined lifecycle process. The implementer must manage and perform tasks in these phases.

- Acquire the software components.
- Prepare the environment
- Assemble the application
- Deploy and start the application
- Perform day-to-day monitoring to makes sure the application is running properly
- Apply code fixes to the application



The RIB supports and follows the RIB Software Lifecycle Management. The RIB Software Lifecycle Management is a well-defined process life cycle and has implemented specific tools and functionality for each of these phases.

- Preparation Phase - In this phase all relevant components are downloaded, extracted configured, and version compatibility checks done.
- Application Assembly Phase - In this phase, site specific configuration changes are made and all the relevant rib-<app>.ears are generated.
- Deployment Phase - In this phase, using the rib-<app>.ears created in the previous step and the site specific information present in a global configuration file, the rib-<app> .ears are deployed to the application server instances.
- Operations Phase - In this phase day-to-day operations of the rib-<app> applications are performed.
- Maintenance Phase - In this phase, code fixes, patching and configuration changes and maintenance of the RIB is performed.

Centralized Configuration and Management

Another key concept in the design of RIB is that all configuration and management is from a single centralized location using specific RIB provided tools. The RIB is built on a completely de-centralized model. However, to ensure consistency and compatibility within an enterprise deployment, a centralized management and configuration model has been designed.

The RIB provides a RIB installer, consistent with all of the Oracle Retail applications, in addition to a command line set of tools that are used at installation, assembly and deployment time to create the Oracle Retail application specific integration. Collectively these command line tools are called the rib-app-builder and provide functionality to support the RIB Software Life Cycle.

Physical Location Considerations

The Oracle Retail Merchandising System (RMS) is the most important core business application from the suite of Oracle Retail Product offerings. RMS provides most of the retail business functionality that Oracle Retail offers its customers. In other words RMS is the central hub of oracle retail applications. Since RMS is the central hub of retail information/data and most information/data flows outward from RMS to other edge retail applications through RIB the decision on where to physically/logically locate RIB is very important and will have direct impact on functioning of your enterprise.

It is recommend to keep the RIB's JMS server logically (not physically) close to the RMS database server as 80% of the data flowing through RIB will interact with RMS database server. Normally RMS up or down status defines your overall enterprise retail business status and so keeping your integration infrastructure status in sync with RMS is beneficial.

RIB Hospital functionality has been added to the TAFR adapters in this release. In order to avoid situations where entire integration can be down just because the TAFR RIB Hospital database is down, it is strategic and beneficial to put the TAFR RIB Hospital tables in the same database instance as the RMS database instance. Obviously it is required to separate the RMS RIB Hospital tables and the TAFR RIB Hospital tables by installing then in their own respective database schemas.

The argument above can be extended to the rib-tafr.ear application and rib-rms.ear application, and so it is recommended to collocate rib-rms.ear and rib-tafr.ear as much as possible.

RWMS and SIM are edge retail applications which might be running closer to your physical warehouse location or your physical store management location. It is recommended collocate rib-sim.ear near SIM application and rib-rwms.ear near RWMS application.

The integration message flow is centrally managed in this release. The rib-func-artifact.war web application determines which messages go where between the rib adapters across all rib-<app> applications. At runtime the rib-<app>.ear needs access to the central message flow repository available in rib-func-artifact.war. Therefore, rib-func-artifact.war must be deployed in a central location where all rib-<app>.ears have access to it at runtime.

The RIB is a central office enterprise integration solution; it is not designed to work optimally on a low (non LAN) bandwidth network. Distribute the rib-<app>.ear applications in such a way where you can avoid lots of network hops, any network protocol bridges, and any communication over a WAN.

JMS Server Considerations

Retail business generates huge volume of transactions that are time sensitive in nature. For the business to be agile and react quickly RIB has to transmit the business events over the JMS server very quickly. The RIB depends upon the underlying JMS server for its performance, robustness, and reliability. Therefore, your retail business performance and reliability is directly dependent on how robust the JMS server is and how much CPU, memory, network and other system resources are available to it. It is critical to provide adequate hardware resource to the JMS server in order for it to be able to meet your performance requirements.

It is not recommended to locate the JMS server and the RIB application server on the same machine. RIB tools automatically configure the JMS server to meets RIB's required configuration. Do not modify the RIB JMS server configuration unless it is advised by RIB documentations. RIB provides tools to monitor the RIB JMS server and only those recommended tools must be used for your daily operations.

It is important to consider the sizing, either file system space or database table space, when planning the deployment of the JMS Provider to a host. It is a very common operational use case for one of the Oracle Retail subscribing applications to go off-line for an extended period, either due to business requirements or problems. Basic sizing at a customer for any JMS system is for the disk (mount points or database) to be able to handle 24 hours of maximum messages per topic.

Using Multiple JMS Servers

Having multiple JMS servers can improve overall system performance and accommodate the following:

- the separation of high volume families from low volume ones.
- the customization of integration flows.
- Operational Quality of Service (QoS).
- distribution of the overall load on the integration system.

To meet the JMS agnostic requirement for the RIB, a unique JMS server ID (jms-server-id) is assigned to each RIB adapter. Accordingly, each RIB adapter can identify the JMS server to which it is associated. As the default, "out-of-the-box" adapters are configured to be on jms-server, jms1.

For each new jms-server-ID, a new resource adapter must be configured to point the application server to the JMS provider's resource. The adapter communicates with the JMS server and is deployed as part of the application. Where customization is required, the adapter can be configured to point to a different JMS.

Note: For more information on using multiple JMS, "JMS Provider Management" in the *Oracle Retail Integration Bus Operations Guide*.

Oracle Streams AQ JMS

Streams AQ provides PL/SQL APIs to interact with the native AQ server inside the Oracle database. Native AQ stream is not the same as AQ behaving as a JMS server. RIB configures the native AQ server to behave as a JMS specification compliant JMS server implementation. Therefore, it is strictly prohibited to manipulate RIB's JMS topics and RIB's AQ configurations directly with the AQ PL/SQL or java API.

AQ JMS server can be configured to be highly available by taking advantage of Real Application Cluster (RAC) functionality of the Oracle Database.

The RIB installation process defines the minimum RDBMS permissions and role that are required for the RIB code to properly create the AQ JMS topics per the specifications required for the RIB behavior. There should be no attempt to use alternate settings or configurations.

Beyond the installation, there are critical considerations that must be addressed for performance and operations that depend on the volumes and topology of the deployment.

Note: See "The RIB on AQ JMS" in the *Oracle Retail Integration Bus Operations Guide*.

For information, see Oracle's documentation around High Availability.

High Availability Considerations

As businesses are maturing and having to do everything quicker, better, faster, and with less resource and money they are pushing similar expectation onto their IT infrastructure. Business users are expecting more out of their IT investments, with zero down time, consistent predictable responding systems which are highly available has become basic requirements of today's business applications.

Modern business application requirements are classified by the abilities that the system must provide. This list of abilities such as availability, scalability, reliability, scalability, audit ability, recoverability, portability, manageability, and maintainability determine the success or failure of a business.

With a clustered system many of these business requirement abilities gets addressed without having to do lots of development work within the business application. Clustering directly address availability, scalability, recoverability requirements which is very attractive to a business. In reality though it is a tradeoff, clustered system increases complexity, is normally more difficult to manage and secure so one should evaluate the pros and cons before deciding to use clustering.

Oracle provides many clustering solutions and options, the ones that are directly relevant to RIB are Oracle database cluster (RAC) and Oracle Application Server clusters.

Oracle Database Cluster (RAC) Concepts

A cluster comprises multiple interconnected computers or servers that appear as if they are one server to end users and applications. Oracle Database Real Application Clusters (Oracle RAC) enables the clustering of the Oracle database. Oracle RAC uses Oracle Clusterware for the infrastructure to bind multiple servers so that they operate as a single system.

Single-instance Oracle databases have a one-to-one relationship between the Oracle database and the instance. Oracle RAC environments, however, have a one-to-many relationship between the database and instances. In Oracle RAC environments, the cluster database instances access one database. The combined processing power of the multiple servers can provide greater throughput and scalability than is available from a single server. Oracle RAC is the Oracle database option that provides a single system image for multiple servers to access one Oracle database. In Oracle RAC, each Oracle instance usually runs on a separate server.

Oracle RAC technology provides high availability and scalability for all database applications. Having multiple instances access a single database prevents the server from being a single point of failure. Oracle RAC enables capability to combine smaller commodity servers into a cluster to create scalable environments that support mission critical business applications.

Note: For information, see Oracle Real Application Clusters documentation.

rib-<app> application and Oracle Database Cluster (RAC)

In this release rib-<app> uses Oracle Streams AQ as the JMS provider. Oracle Streams AQ is built on top of an Oracle database system. Since AQ is hosted by Oracle database system, RIB can take advantage of database RAC capability for its JMS provider. By using RAC configured AQ as the RIB's JMS provider you can scale the RIB's JMS server vertically and horizontally to meet any retailer's scalability and high availability need.

At runtime rib-<app> uses the database for keeping track of its RIB Hospital records. These RIB Hospital tables can be hosted by an Oracle RAC database providing high availability and scalability for these RIB Hospital records.

Oracle Application Server Cluster Concepts

Oracle Application Server cluster is defined as two or more Oracle Application Server instances loosely connected to each other providing high availability and scalability for hosted business applications.

Any highly available system has to have redundant components to mask failures in individual components. All Oracle Application Server components can be deployed in a redundant fashion to make their services more available. Oracle Application Server allows choosing between active-active or active-passive redundant models in all its sub-tiers.

In an active-passive configuration the passive components are only used when the active component fails. Active-passive solutions deploy an active instance that handles requests and a passive instance that is on standby. In addition, a heartbeat mechanism is usually set up between these two instances together with a hardware cluster (such as Sun Cluster, Veritas, RedHat Cluster Manager, and Oracle CRS) agent so that when the active instance fails, the agent shuts down the active instance completely, brings up the passive instance, and resumes application services.

In an active-active model all equivalent members are active and none are on standby. All instances handle requests concurrently.

Obviously, an active-active system generally provides higher transparency to consumers and has a greater scalability than an active-passive system. On the other hand, the operational and licensing costs of an active-passive model are lower than that of an active-active deployment.

For those systems requiring an active-active model, Oracle Application Server provides Oracle Application Server Clusters, a set of application server instances configured in an active-active model to serve the same set of applications and/or services. When an active-passive model is needed, Oracle Application Server provides Oracle Application Server Cold Failover Cluster, which is a set of application server instances (two in most cases, since only one remains active and no greater benefits are achieved by including more nodes) configured in an active-passive model to serve the same set of applications and/or services.

Oracle Application Server allows the grouping of Java EE containers that serve the same application through Oracle Application Server Cluster (Oracle Containers for Java EE). The connectivity provided within a cluster is a function of Oracle Notification Server (ONS), which manages communications between Oracle Application Server components, including OC4J and Oracle HTTP Server. The ONS server is a component of Oracle Process Manager and Notification Server (OPMN), which is installed by default on every Oracle Application Server host. When configuring a cluster topology, you are actually connecting the ONS servers running on each Oracle Application Server node which manages the system handshake and coordination.

Note: For information, see the Oracle High Availability documentation.

---

rib-<app> application and Oracle Application Server Cluster

RIB uses a JMS server for message transportation between the integrating retail applications. Since RIB must preserver the message publication and subscription ordering, rib-<app>s deployed in Oracle Application Server cannot be configured in an active-active cluster mode. In active-active cluster mode, multiple subscribers and publishers process messages simultaneously and there is no way to preserve message ordering.

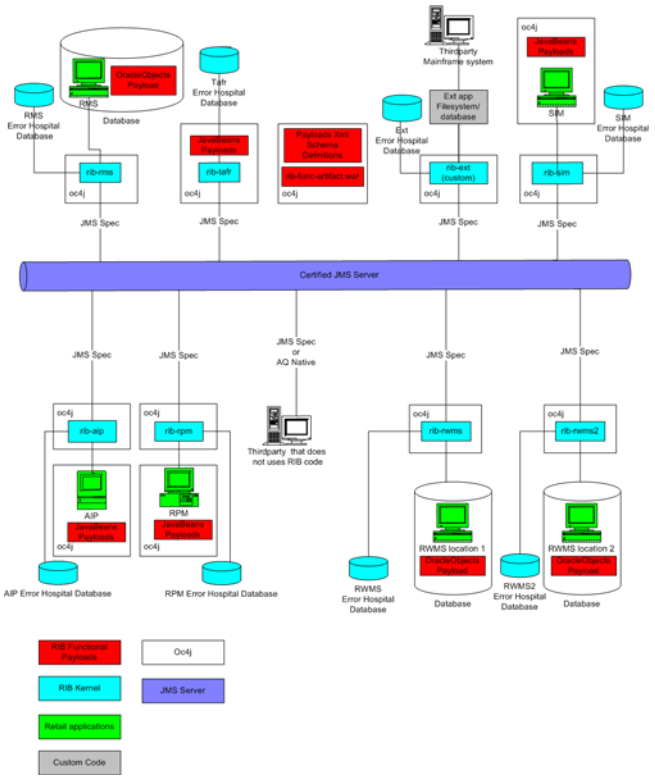
rib-<app> can be deployed to a "single" oc4j instance of an Oracle Application Server that is clustered(active-active). In this configuration even though rib-<app> is deployed in an OAS cluster, multiple instance of same rib-<app> is not running at the same time as there is only one oc4j instance where the rib-<app> is deployed and so RIB can still preserve message ordering. The maximum number of JVM (Java Virtual Machine) hosting a rib-<app> oc4j instance must always be configured to be 1 for the same reason of preserving message ordering.

To truly configure rib-<app>s for high availability the only option is to configure it in active-passive mode.

6

Deployment Architecture and Options

As the logical architecture diagram shows, there are many rib-<app> applications that coordinate message flows between the various Oracle Retail applications. There are no physical location constraints on where these rib-<app> applications can be deployed as long as they are visible from the same network. But the decision on where to physically and logically locate your rib-<app> applications has a huge impact on the high availability, performance and maintainability of your integration solution, so this decision must be given careful consideration.



Recommended Deployment Options

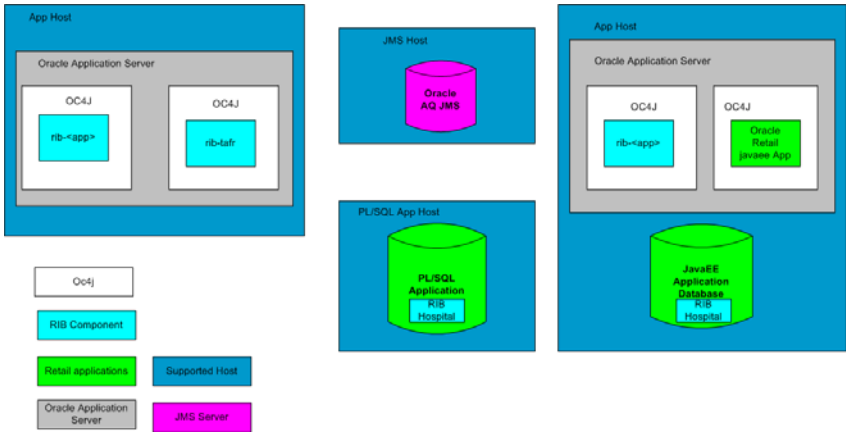
The RIB applications can be deployed in a variety of physical and logical configurations depending on the retailer's needs. Oracle Retail has two recommended configuration alternatives.

- Distributed In this deployment each of the rib application (rib-<app>.ear) is deployed in the same Oracle Application Server as integrating application (<app>.ear) but in its own oc4j instance.
- Centralized In this deployment all rib applications (rib-<app>.ear) are deployed in a single Oracle Application Server (not oc4j instance) independent of where the Oracle Retails apps (<app>.ear) Oracle Application Server is.

In all cases, the rib application (rib-<app>.ear) should be deployed in its own OC4J Instance. It is not recommended to deploy multiple rib applications into the same OC4J instance, or to have the rib application (rib-<app>.ear) deployed into the same OC4J instance as the integrating application (<app>.ear).

Distributed Deployment Alternative

In this deployment setting the rib application (rib-<app>.ear) is deployed in the same Oracle Application Server as the integrating application (<app>.ear). Logically rib-<app>.ear and <app>.ear are closely tied to each other, so it makes sense to also deploy them physically close to each other within the same Oracle Application Server. There will be only one JMS server and all participating rib-<app> are configured to use the same JMS server.



Following are some advantages and disadvantages of this configuration.

Advantages

- Required single Oracle Application Server for both rib (rib-<app>.ear) and integrating application (<app>.ear).
- <app>.ear and rib-<app>.ear are close to each other but are still loosely coupled.
- It is easy to find which rib-<app>.ear is associated with which integrating application (<app>.ear).

- A single OAS instance is never the single point of failure for the whole integration system.
- The Oracle Process and Notification manger can optimize network communication much better when both the oc4j instances are managed by the same OAS instance.

Disadvantages

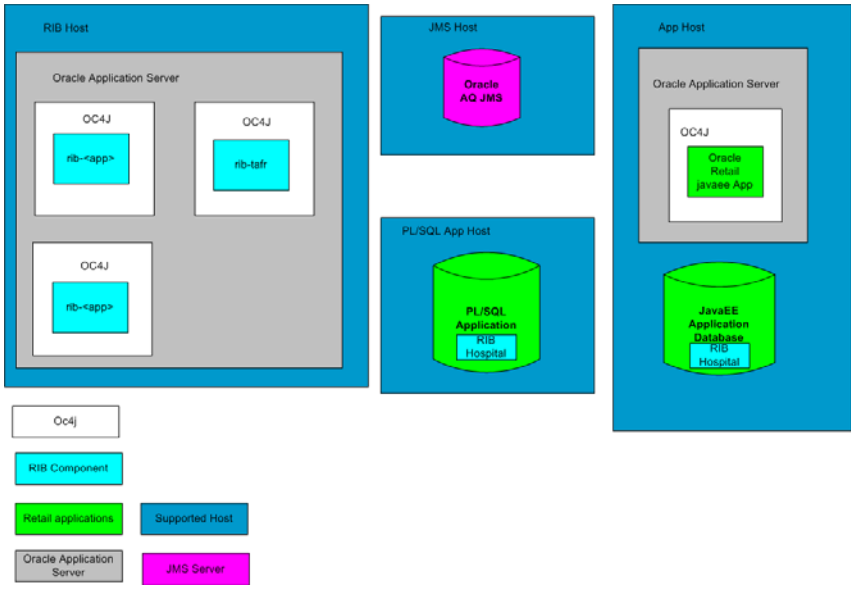
- When OAS server of rib-<app>.ear has to be bounced, the integrating application (<app>.ear) becomes unavailable as both reside in the same application server. Similarly rib-<app>.ear has to bounce when <app>.ear needs bouncing. This dependency between the two applications is not ideal.
- Even though both the applications reside within the same application server, it is the configuration with the applications that are tying them together not the physical characteristics of both being deployed in the same application server. Physical location might be misleading if the system is not configured correctly.
- One application server has to work harder for management of resources and services for both applications.
- System load distribution between rib-<app>.ear and <app>.ear is not possible as both applications reside within the machine.

Who Should Use This Configuration?

Medium to large size deployments can use this configuration. This configuration is appropriate when the machine hosting OAS is adequately sized for its job. A high message volume in rib-<app>.ear can adversely affect the performance of the integrating application (<app>.ear) in areas that are not related to integration. Ideally this kind of behavior is not desirable for an online system.

Centralized Deployment Alternative

In this deployment all rib application (rib-<app>.ear) are deployed in a single Oracle Application Server but in separate Java EE containers (OC4J Instances). The integrating applications (<app>.ear) are deployed in their own separate Oracle Application Server. There is only one JMS server and all participating rib-<app> are configured to use the same JMS server.



Following are some advantages and disadvantages of this configuration.

Advantages

- All integration relegated components are deployed in one application server.
- Simple to find, view, and manage.

Disadvantages

- Since all rib-<app>.ear resides in the same Oracle Application Server, system resources get shared between the applications which means that they can adversely affect each others performance. For example rib-aip.ear can become slow when rib-rpm.ear is processing lots of messages even though these applications are not at all related to each other.
- Overall performance can be slower as one application server machine has to do lots of work.
- The RIB application server and host become the single point of failure for the whole integration system (environment). That is, when the Oracle Application Server goes down the whole integration is down for all retail applications (<app>.ear).

Who should use this Configuration?

Small to Medium size deployments can use this configuration. When the message size is small and high volume is not expected this configuration can be used. This configuration can also be used when there are only two integrating application. As each rib-<app>.ear publishes and subscribes to each other, they are indirectly (through JMS) interdependent and so performance should not be affected too much when the message volume is less.

Conclusions

RIB deployment recommendation does not take into account your hardware size, network topology, existing legacy system, and so on. One size fits all does not work. You need to do proper due diligence based on our recommendations and your specific environment settings in order to come up with the best deployment architecture that meets your needs.

7

Implementation Process

This release of RIB defines the full lifecycle of the RIB software product. The RIB lifecycle and phases are described in detail in the software lifecycle management section of this document. For every lifecycle phases and task that RIB defines it provides corresponding tools and utilities to manage and operate on those phases. The tools and utilities are described in detail in the RIB Operations Guide.

There are several prerequisite steps that should be followed to have a successful RIB installation and deployment.

- Understand the RIB Core Concepts.
- Understand the integration message flow paths.
- Understand the deployment options.
- Understand the RIB Life Cycle.
- Understand the physical and logical requirements and limitations of the RIB Components.
- Understand the RIB Operational considerations.

The process of implementation should follow these general steps:

- Work with the teams at your organization dedicated to Oracle Retail to coordinate plans for the number and type of environments needed (for example, Dev, Integration, Production).
- Each type of environment needs to be sized, deployed, and managed in conjunction with the implementation of the Oracle Retail applications.
  - It is important to understand the volume requirements of the production system so that the appropriate decisions can be made about the deployment option and the physical location and sizing.
- All deployments have integration to existing retailer systems. It is critical to understand the position of the RIB as it fits into the overall integration architecture and that the current operations and architecture team understand the RIB and its capabilities.
- Select a deployment option (centralized or distributed),
  - This may be mixed depending on the phases of deployment. Development and test may be centralized and production distributed.
  - Understand the operational complexities of each and plan for the staffing.
- Work with the application server administration team(s) to determine the physical and logical placement of the RIB components.

- Work with the system administrator and database administrator to appropriately place, size, and configure the AQ JMS.
- Work with the system administrators to select the central RIB management location(s); rib-home.
- The installation of the RIB has many pre-requisites and dependencies that require the understanding, support and effort of database administrators, system administrators, application server administrators, and your organization's Oracle Retail application teams. It is a critical role of the RIB system administrator to work with each team, regardless of the site organization structure. See the *Oracle Retail Integration Bus Installation Guide*.
  - The operation requirements and considerations are covered in the *Oracle Retail Integration Bus Operations Guide*. The guide should be understood before the implementation so that the factors can be considered in the planning.
- Create operational plans for the RIB Life Cycle. See the *Oracle Retail Integration Bus Operations Guide*.
- Create plans for environment monitoring and maintenance. See the *Oracle Retail Integration Bus Operations Guide*.
- Plan to performance test. The RIB supplies tools to aid in the testing, but it is a difficult task that involves the database administrators, system administrators, application server administrators, and the Oracle Retail application teams.

Note: For more discussion on Performance see "Performance Considerations" in the *Oracle Retail Integration Bus Operations Guide*.

Implementation Verification and Validation

Verification is the process of reviewing, inspecting, testing, and documenting that the product behaves in a manner as defined by the product requirement specification. Validation on the other hand is the process of making sure that the product's runtime behavior meets the retailer's needs and requirements. RIB provides tools and utilities to verify that a RIB installation is configured correctly and works properly when business events (messages) occur in your enterprise. RIB also provides tools to test your integration infrastructure standalone independent of any Oracle Retail applications.

Implementation Environment Verification

The RIB Diagnostic and Monitoring Tool (RDMT) can be used to verify your installation and configurations. The RDMT configuration report utility generates an extensive configuration report of your runtime environment. It is recommended to regularly perform full RIB health check using the RDMT tool sets to proactively find problems and recover before any problem becomes a serious issue.

See the *Oracle Retail Integration Bus Operations Guide* for RDMT information.

Integration Environment Testability

Identifying the ownership of an integration problem is one of the hardest problems in any integration project. Data mismatch problems always show up in the integration layer but in reality it is the source and the destination applications that has mismatch data model. To be able to isolate integration infrastructure problem versus retail application API problem it is very important to be able to test the integration infrastructure independent of the retail applications.

In this release, RIB provides two test harnesses that allow you to build a standalone working integration environment without the need to install any Oracle Retail applications. The test harnesses simulate Oracle Retail PL/SQL applications (RMS and RWMS) and Oracle Retail Java EE applications (SIM, RPM and AIP). The test harnesses are known as plsql-api-stubs and javaee-api-stubs respectively.

See the *Oracle Retail Integration Bus Operations Guide* for further detail on the RIB test harness.

8

Performance

Performance Factors

The performance of each of these components is influential in the overall performance of the system:

- The application server(s) topology and configuration.
- The RIB deployment approach.
- The hardware sizing and configuration of the RIB hosts.
- The hardware sizing and configuration of the applications that are connected to the RIB.
- The hardware sizing and configuration of the JMS provider host.
- The hardware sizing and configuration of the RIB Hospitals hosts.

There are other factors that determine the performance of the overall system. Some of these factors in a RIB environment are:

- Number of channels configured
- Number of messages present in the topic
- Size of the message
- Database clustering
- Application Server topology
- Number of TAFRs in the processing of the message
- Message aggregation

See the "Performance Testing the RIB" in the *Oracle Retail Integration Bus Operations Guide*.

Performance and Parallel Logical Channels

The RIB has to provide guaranteed once and only once processing of business events (message) across the enterprise. Maintaining the ordering of business events across the enterprise is very important from the data integrity point of view.

To provide guaranteed sequencing of message processing RIB requires a guaranteed first in, first out (FIFO) messaging system with guaranteed FIFO rollback. That is, when you rollback the message from the consumer you get the same message back the next time so that it is processed in sequence. JMS Provider provides this FIFO topic and FIFO rollback capability that enables RIB to guarantee message sequencing.

Processing messages in sequence has some overhead as every message has to be checked against the database to find the status of previous messages that it is dependent (same businessObjectid") on. Sequencing requirement creates an inherent bottleneck of ability to process only one message at a time. For example, messages can come at the rate of 100 messages per second but RIB subscribing adapter can process only 1 message at a time in order to preserver ordering. To get around this bottleneck and improve performance RIB provides a few optimizations and functions.

First, RIB processes messages in sequence only when the publishing application wants it to be processed in sequence. The message producer application defines a businessObjectid whose existence informs RIB that this and all subsequent messages with the same businessObjectid have to be processed in order.

Second, parallel logical channels can be created for each message flow paths in the integration system to improve performance. Parallel logical channels are virtual logical message flow paths within the same physical JMS topics. To add additional channels each adapter participating in a message flow must be configured with additional adapter instances. See the *Oracle Retail Integration Bus Operations Guide* for how to configure parallel logical channels.

Parallel logical channel is not the solution for all of your performance problems in the integration system. Parallel logical channels can help only when the corresponding applications integration API is written with non-locking logic and concurrency invocation in mind.

Generally the retail application's integration APIs are the biggest factor and bottlenecks in the overall messaging system through put. It is not appropriate to start creating parallel logical channels at the first sign of performance problem. It is critical to analyze your retail applications integration APIs first and tuning them before considering the creating of parallel channels.

Parallel logical channel comes with an increase in complexity, more CPU demands and memory requirements, and more operational overhead. They should only be considered when all other components are fully tuned and you are still not able to meet your target numbers.

9

Security

Security in the integration layer is a big concern for every retail enterprise. The security system should be open enough to allow trusted remote applications to integrate easily and, at the same time, lock down unauthorized remote access. To address security concerns RIB utilizes the security modules available in the Oracle middle ware and database systems.

There are two categories of administrators in RIB: RIB System Administrators and RIB Application Administrators. The RIB System Administrators are involved in installing, configuring, deploying defect fixes, and making sure that the integration infrastructure is up and running properly. The RIB Application Administrators are the people who are mostly concerned with the business side of the integration system. These users (RIB application administrators) mostly bring up or down RIB adapters and fix data issues with message payloads using RIHA. There are different realms, roles and users defined for each category of RIB administrators.

RIB Application Administrators Security Domain

For each rib-<app>.ear deployed, RIB creates a security realm called ribadminrealm. This realm defines a role called ribadminrole. By default, RIB creates a user called ribadmin that belongs to the ribadminrole. The default password for ribadmin user is ribadmin. The RIB System Administrators can manage rib-<app> application's users and access control through the Oracle Application Server Control (em). The default realm, role and user that RIB creates must not be deleted or modified.

RIB System Administrators Security Domain

The RIB System Administrators mainly focus on managing access the RIB's JMS server, application server instances, RIB Hospital database, and the rib-home workspace. RIB can be deployed with a user other than the OAS default oc4jadmin user, to configure a riboc4jadmin user see the *Oracle Retail Integration Bus Installation Guide*.

10

Integration with Fusion Middleware

The RIB is certified on the Oracle Fusion Middleware Application Server. All of the RIB publishers, subscribers, and TAFRs are JavaEE standard components (EJBs and MDBs) that are deployed and managed by the Oracle Application Server in OC4J instances. This means that the RIB can be deployed into an existing Fusion Middleware architecture without any changes.

All RIB message payloads are fully standard compliant XSD based. All of the XML payloads are namespace aware and follow the general standards as well as the conventions that make them compatible with other Oracle Fusion products such ESB and BPEL. The payload schema definitions (xsds) are packaged with each release along with sample messages.

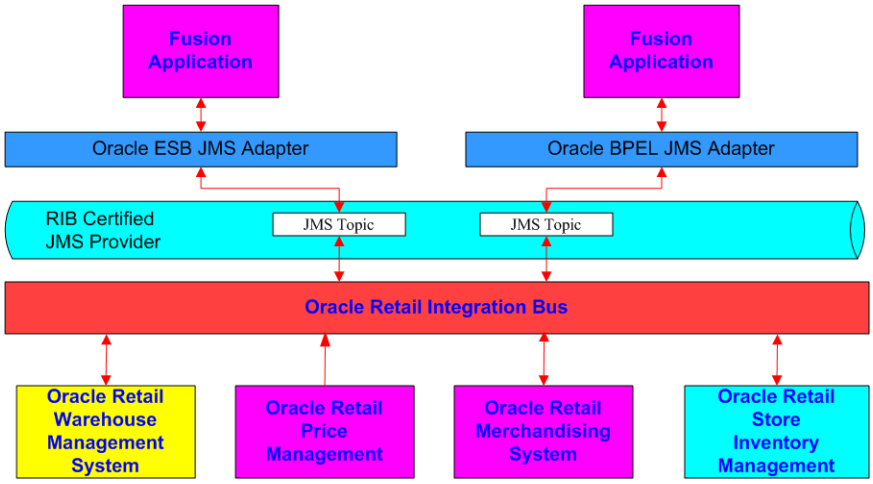
The RIB is a standard's based JMS messaging system, with Oracle Streams AQ and the Oracle Application Server OC4J JMS being the initial JMS providers certified.

The recommended approach for integration between the RIB and Oracle Fusion Middleware products is at the JMS topic level. Any standards compliant tool or product that can interface to the JMS and subscribe and publish messages can be integrated with the RIB.

There are some key functional requirements that an integrating application must follow:

- Ability to connect to a standard JMS and publish to a topic.
- Ability to create a durable subscriber to a RIB JMS topic
- Ability to set user-defined message properties.
- Ability to encode and decode RIB payloads embedded within the RIB message envelope.

General RIB to Fusion Middleware Architecture



The Oracle Fusion Middleware products, such as ESB and BPEL, use a common standard JMS Adapter. This adapter can be used to connect to the RIB certified JMS Provider and topics.

The JMS topics that the RIB creates for Publication and Subscription are detailed in the Oracle Retail Integration Guide, along with all of the message payloads for each message family.

The RIB html encodes each message payload and inserts it into the RIB messages envelope. Each message has a JMS user-defined property called threadValue that is required to be set on all in-bound messages. In a multi-channel message flow, the subscriber will need to set the message selector to an appropriate threadValue to maintain message publication sequencing.

The xml schema definitions for the payloads and the RIB Messages envelopes are packaged and shipped with the RIB.

See the *Oracle Retail Integration Bus Integration Guide* for information.

The RIB JMS topic names and message flows between the RIB adapters for each of the Oracle Retail applications are defined in the rib-integration-flows.xml file. This file is the single source of truth that the RIB release uses at configuration and run-time. It is required to be accessible within each RIB deployment: http://<server>:<port>/rib-func-artifact/rib-integration-flows.xml. During installation and configuration, this file is deployed as part of the functional artifact war file.

General Process of Integration

The general process for custom integration with the RIB:

- Determine the Message Family of interest (e.g., Items)
- Use the RIB Integration Guide to determine the message payloads and topics involved.
- Configure the JMS Adapter within the tool (ESB/BPEL) to the RIB JMS provider.
- Understand the RIB envelope (RibMessage.xsd) and the message type relationship.

- Understand the payload for each message. These are html-encode inside the RibMessage envelope.
  - The RIB XSDs are included in the RIB Integration Guide as well as the Function Artifacts war file.
- Understand the Oracle Retail Application API mappings. These are included in the RIB Integration Guide. This is important because the XSDs do not reflect the actual optional/mandatory state of an element. For historical reasons (to support previous releases) all elements in the XSD that have been added since RIB version 10.3 have been optional at the message level.
  - The Mapping reports are included with the Integration Guide.
  - Each of the Oracle Retail Applications has documentation on the behavior of the API.
- All RIB messages must have the message property threadValue set by publishing applications, and in a multi-channel message flow, the subscriber will need to set the message selector to an appropriate threadValue to maintain message publication sequencing.
  - Understand the relationship between the threadValue and multiple-channels within the RIB. See "Multiple Channels" in the *Oracle Retail Integration Bus Operations Guide*.
- Many of the Message Families have a RIB Component called a TAFR involved. Understand what a TAFR is and how it works within a message flow. This can be very involved in some families, and can actually create additional mandatory elements with a message that may not be obvious. See "Transform, Filtering and Routing" in the *Oracle Retail Integration Bus Operations Guide*.
- The RIB Integration Guide for each family has the general functional specifications for the TAFRs involved with that family.
- Understand the volume characteristics of a message family. The RIB is designed to handle retail volumes, so a poorly designed subscriber can have a huge impact on the JMS. Conversely, a publisher that tries to use the RIB as a bulk transfer mechanism is also inappropriate.

Example - Configure FWM JMS Adapter to RIB AQJMS

This example demonstrates configuring a simple JMS Adapter connection to a RIB AQ JMS. This example assumes the GA installation of the Oracle SOA Suite and the default directories ($SOA_HOME).

To do the configurations, it is necessary to have the RIB AQ JMS information:

AQ JMS Database URL	jdbc:oracle:thin:ribaq/ribaq@linux1:1521:ora10g
AQ Username	ribaq
AQ Password	ribaq

Create the Resource Provider

First edit the $SOA_HOME/oc4j_soa/config/ application.xml file to add a resource provider that points to the RIB AQ JMS. Provide a name to the resource provider.

```
<resource-provider class="oracle.jms.OjmsContext" name="ribaq">
    <description>RIB AQ</description>
    <property name="url"
        value="jdbc:oracle:thin:ribaq/ribaq@linux1:1521:ora10g" />
    <property name="username" value="ribaq" />
    <property name="password" value="ribaq" />
</resource-provider>
```

Configure a JMS Connection Factory

Edit the $SOA_HOME/oc4j_soa/application-deployments/default/JmsAdapter/ oc4j-ra.xml file to match the Resource Provider name created above. Either create a new entry or edit an existing one. Note the factory location (JNDI Name).

```
<connector-factory location="eis/aqjms/Topic" connector-name="Jms Adapter">
    <config-property name="connectionFactoryLocation"
        value="java:comp/resource/ribaq/TopicConnectionFactories/myTCF"/>
    <config-property name="factoryProperties" value=""/>
    <config-property name="acknowledgeMode" value="AUTO_ACKNOWLEDGE"/>
    <config-property name="isTopic" value="true"/>
    <config-property name="isTransacted" value="true"/>
    <config-property name="username" value="ribaq"/>
    <config-property name="password" value="ribaq"/>
    <connection-pooling use="none">
    </connection-pooling>
    <security-config use="none">
    </security-config>
</connector-factory>
```

Configure the FMW JMS Adapter

There is nothing special about configuration of the JMS Adapter in either ESB or BPEL to now connect to the Resource Provider configured to the RIB AQ JMS. (See the Oracle SOA Suite tutorials and documentation).

The following prompts appear during configuration:

Prompt	Example
Resource Provider	ribaq
JNDI Name	eis/aqjms/Topic
Connection (database)	ribaq/ribaq@linux1:1521:ora10g
Destination Topic (Browse to see RIB Topics)	java:comp/resource/ribaq/Topics/RIBAQ.ETORDERFROMRMS
Message Selector	threadValue=1
Durable Subscriber ID:	Order_sub_ESB
Schema Location	RibMessages.xsd
Schema Element	RibMessages

Note: RibMessages.xsd should be imported into the project from the RIB functional artifacts distribution.

11

RIB Customization/Extension

The "customization" of an Oracle Retail application often drives requirements to customize or extend the messages that flow among the Oracle Retail applications, or to create new message flows to support new business logic.

This document discusses the customization/extension approaches and best practices from an Oracle Retail Integration Bus (RIB) perspective for extending base messages, and for creating new messages and adapters. These are complex topics and should be performed with great care to avoid making future generally available (GA) releases difficult or impossible to accept.

Retailers often modify retail software either in-house or through third party system integrators. The customization and extension of Oracle Retail base products and messages are not supported by Oracle Retail, including My Oracle Support. This document aims to mitigate the risks of unsupported customization by providing guidance and references on how to attempt to customize safely and effectively. The tools and approaches described in this document are complex and require a high level of skill and knowledge of the product. Any issues that may arise with custom flows, custom APIs, or customized message families are the responsibility of the customer and not Oracle Retail.

Prerequisites

Customization requires a number of considerations and planning steps. Planning helps prevent reinstallation and rearchitecture of the RIB due to operational or performance problems.

- A functional RIB environment without customizations.
- Familiarity with core RIB concepts, components, and architecture, including an understanding of all the following:
  - Oracle database triggers, RIB adapters, RIB Message envelope, RIB message payloads, and the functionality of GETNXT () and CONSUME () stored procedures.
  - The integration message flow paths.
  - The RIB life cycle.
  - The physical and logical requirements and limitations of the RIB components.
  - RIB operational considerations.

The tools used in the customization and extension of the RIB are documented separately. The primary tools are the Oracle Retail Functional Artifact Generator and the rib-app-builder tools.

General Customization Rules

- Always keep an environment with a base version release to reproduce any base version issues. Only GA base code and messages are supported.
- Always take a backup of the files being modified during the customization to allow for reversal of the changes.
- As often as possible, use RIB tools such as RDMT, RIHA, and "Stubby" to test the customized changes.
- Never modify the existing base flows in rib-integration-flows.xml. Modification can cause errors in functionality that are difficult to detect. Also, modifications made to base flows do not carry over to new releases, nor are they retained when defect fixes are applied to base code and objects.
- When customizing or extending RIB messages or flows, all publishing and subscribing applications that participate in the flow must be considered.
- In scenarios where payload customization or the addition of new message type for a particular message family is planned, and the flow contains a TAFR, the following additional rules apply:
  - TAFRs that do not examine RIB message types or payloads do not require modification.
  - For TAFRs that examine message types or payloads for filtering or transformation purposes, the TAFR implemention code must be changed. If this code is not changed, the messages will fail and land in RIB Error Hospital tables.

Message Family and Message Type Customization

In the RIB, all messages are categorized by "message family" and "message type." A message family is specific to one or more Business Objects. It defines all publishable events occuring on the Business Objects.

The message type classifies a specific event. For example, the Orders message family is for messages about purchase orders, and the Vendor message family is for supplier or vendor information.

Typically, a message family includes at least one Create, Mod, and Delete operation.

Note: See "Message Family and Message Types" in Chapter 3 .

Adding a New Message Type

To add a new message to an existing message family, the simplest approach is to add a new message type. The first step is to determine and create the payload for the new message type. The message payload must be created following the guideline and packaging rule for RIB messages.

Note: To create a new message family XSD, see "Adding a New Payload."

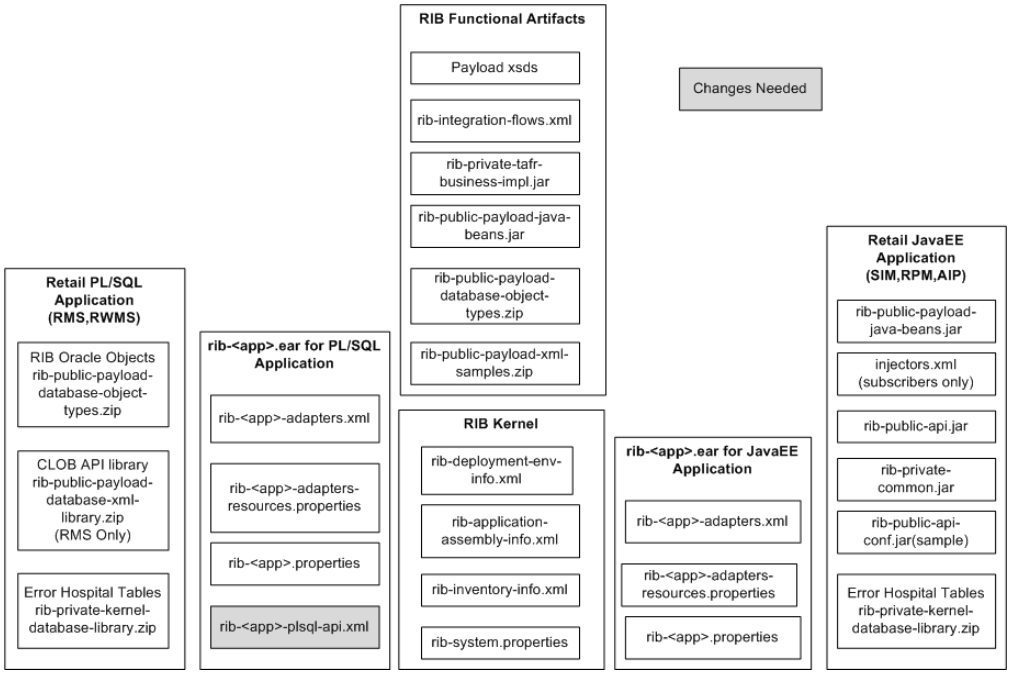
Once the desired payload is ready, follow the instructions in the following sections, depending on the type of applications in the message family and in the message flow.

Message Flows with PL/SQL Applications

The new message type created for an already existing or new message family must be added in the rib-<app>-plsql-api.xml of the subscribing PL/SQL retail application.

> **Note:** No configuration changes are needed in rib-<plsql app> whenever PL/SQL applications publish a new message type to which no PL/SQL applications subscribe.

The following illustration indicates the files that must be changed inside the RIB infrastructure during the addition of new message type when a PL/SQL application is involved in the message flow.



Procedure for Adding a New Message Type for PL/SQL Applications

To add a message type for message flows involving PL/SQL applications, complete the following steps.

1. Add the new message type in rib-<app>-plsql-api.xml, where app = rms or rwms in the <RIB_HOME>/application-assembly-home/rib-<app> directory.

   For example, to add the new message type, "DiffGrpFooCre," for the DiffGrp message family that is subscribed by RWMS, add the message type under the <adaptorClassDef name="DiffGrp_sub"> of rib-rwms-plsql-api.xml in the <RIB_HOME>/application-assembly-home/rib-rwms as shown below.

```
cd <RIB_HOME>/application-assembly-home/rib-rwms
vi rib-rwms-plsql-api.xml

<adaptorClassDef name="DiffGrp_sub">
<class>com.retek.rib.collab.general.OracleObjectSubscriberComponentImpl</class>
                <messageFamily name="DiffGrp">
                        <storedProc>
                                <signature>{call RDMSUB_
DIFFGRP.CONSUME(?,?,?,?,?)}</signature>
```

```
                                <useFacilityType>true</useFacilityType>
                        </storedProc>
                        <messageType name="DIFFGRPDEL">
                                <oracleObject>RIB_DiffGrpRef_REC</oracleObject>
                        </messageType>
                        <messageType name="DIFFGRPDTLCRE">
                                <oracleObject>RIB_DiffGrpDtlDesc_REC</oracleObject>
                        </messageType>
                        <messageType name="DIFFGRPDTLDEL">
                                <oracleObject>RIB_DiffGrpDtlRef_REC</oracleObject>
                        </messageType>
                        <messageType name="DIFFGRPHDRCRE">
                              <oracleObject>RIB_DiffGrpHdrDesc_REC</oracleObject>
                        </messageType>
                        <messageType name="DIFFGRPDTLMOD">
                              <oracleObject>RIB_DiffGrpDtlDesc_REC</oracleObject>
                        </messageType>
                        <messageType name="DIFFGRPHDRMOD">
                              <oracleObject>RIB_DiffGrpHdrDesc_REC</oracleObject>
                        </messageType>
<messageType name="DIFFGRPFOOCRE">
                               <oracleObject>RIB_DiffGrpHdrDesc_REC</oracleObject>
                </messageType>
        </messageFamily>
</adaptorClassDef>
```

Note: Creating a temporary working directory, "customization-workarea," under <RIB_HOME>/tools-home is recommended. This directory can be used when performing customization related tasks.

2. Edit the payload.properties file in the ./conf directory of the Artifact Generator tool installation. The payload.properties contains the new payload message definitions. The format of the definition is as follows:

```
RIBFAMILY.TYPE=IMPLEMENTATION CLASS NAME
cd conf
vi payload.properties (make changes)
```

For example, to add the new message type, "DiffGrpFooCre," for th DiffGrp message family, the modification to payload.properties is as follows:

```
DIFFGRP.DIFFGRPFOOCRE= com.retek.rib.binding.payload.DiffGrpHdrDesc
```

In this case, DiffGrpFooCre calls the implementation class, DiffGrpHdrDesc.

Note: For the maximum supported length of the message type, see the RibMessages.xsd in the rib-func-artifact.war.

Note: If a TAFR is involved in the flow, the appropriate changes must be made to the TAFR to handle the new message types.

3. Run the Artifact Generator to generate various functional artifacts.

   ```
   $GROOVY_HOME/bin/groovy GenArtifacts.groovy
   ```

   Each generated artifact is in the appropriate ./output*/dist folder, including:

   - rib-public-payload-database-object-types.zip
   - rib-public-payload-java-beans.jar
   - retail-public-bo-java-beans.jar
   - rib-public-payload-xml-samples.zip

4. Copy the newly generated artifact rib-public-payload-java-beans.jar from the appropriate ./output*/dist folder to <RIB_HOME>/application-assembly-home/rib-func-artifacts/ directory.

5. Run the rib-app-compiler.sh script from <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and prepare it for deployment.

   Note: See the *Oracle Retail Integration Bus Operations Guide* for information about rib-app-builder tools.

   ```
   cd <RIB_HOME>/application-assembly-home/bin
   sh rib-app-compiler.sh
   ```

6. Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

   ```
   cd <RIB_HOME>/deployment-home/bin
   sh rib-app-deployer.sh -deploy-rib-func-artifact-war
   ```

   This step deploys the rib-func-artifact.war.

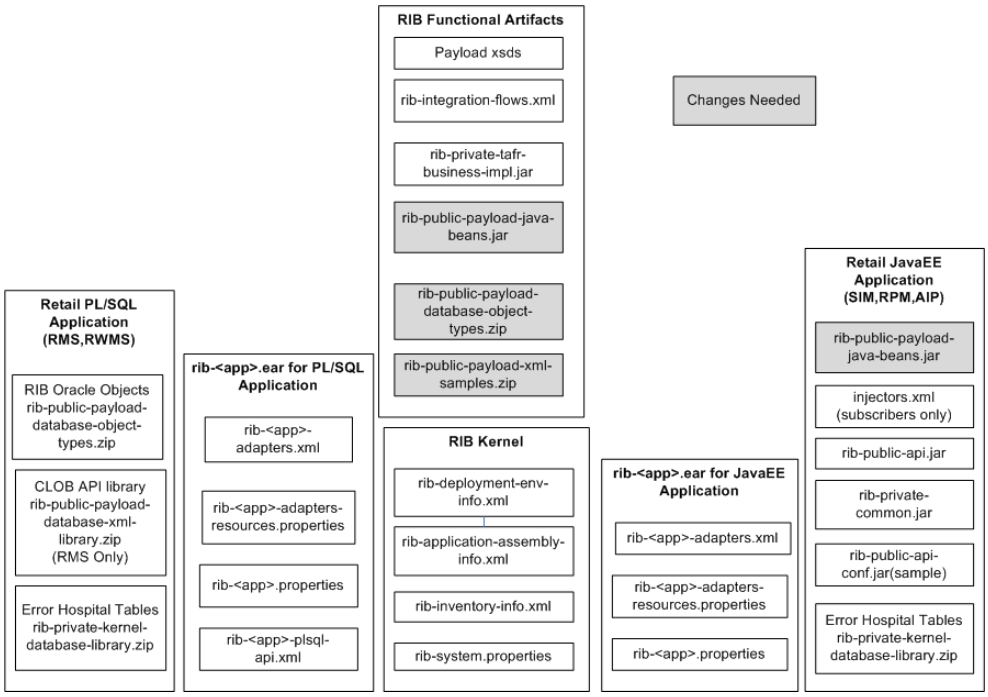
   ```
   sh rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
   ```

   The rib-<app> is deployed to the Java EE container. Repeat this step for each rib-<app> in the integration environment.

   Note: The <app> value must be rms or rwms.

Message Flows with Java EE Applications

The following illustration indicates the files that must be changed inside the RIB infrastructure during the addition of new message type when a Java EE application is involved in the message flow.



Procedure for Adding a New Message Type for Java EE Applications

To add a message type for message flows involving Java EE applications, complete the following steps.

> **Note:** Creating a temporary working directory, "customization-workarea," under <RIB_HOME>/tools-home is recommended. This directory can be used when performing customization related tasks.

1. Edit the payload.properties file in the ./conf directory of the Oracle Retail Artifact Generator tool installation. The payload.properties contains the new payload message definitions. The format of the definition is as follows:

   RIBFAMILY.TYPE=IMPLEMENTATION CLASS NAME

   ```
   cd conf
   vi payload.properties (make changes)
   ```

   For example, to add the new message type, "FooDeptCre," under the Merchhier message family to call the implementation class, MrchHrDeptDesc, the file is modified as follows:

   ```
   MERCHHIER.FOODEPTCRE= com.retek.rib.binding.payload.MrchHrDeptDesc
   ```

Note: For the maximum supported length of the message type, see the RibMessages.xsd in the rib-func-artifact.war.

Note: If a TAFR is involved in the flow, the appropriate changes must be made to the TAFR to handle the new message types.

2. Run the Artifact Generator to generate various functional artifacts.

   ```
   $GROOVY_HOME/bin/groovy GenArtifacts.groovy
   ```

   Each generated artifact is in the appropriate ./output*/dist folder, including

   - rib-public-payload-database-object-types.zip
   - rib-public-payload-java-beans.jar
   - retail-public-bo-java-beans.jar
   - rib-public-payload-xml-samples.zip

3. Copy the newly generated artifacts (listed above) from the appropriate ./output*/dist folders to the <RIB_HOME>/application-assembly-home/rib-func-artifacts/ directory.

4. Run the rib-app-compiler.sh script from <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

   ```
   cd <RIB_HOME>/application-assembly-home/bin
   sh rib-app-compiler.sh
   ```

5. Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

   ```
   cd <RIB_HOME>/deployment-home/bin
   sh rib-app-deployer.sh -deploy-rib-func-artifact-war
   ```

This step deploys the rib-func-artifact.war.

```
sh rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
```

The rib-<app> is deployed. Repeat this step for each rib-<app> applicable to the integration environment.

Note: The <app> value must be rms or rwms.

Note: To verify the addition of a new message type for a message family, see "Verifying the New Message Type."

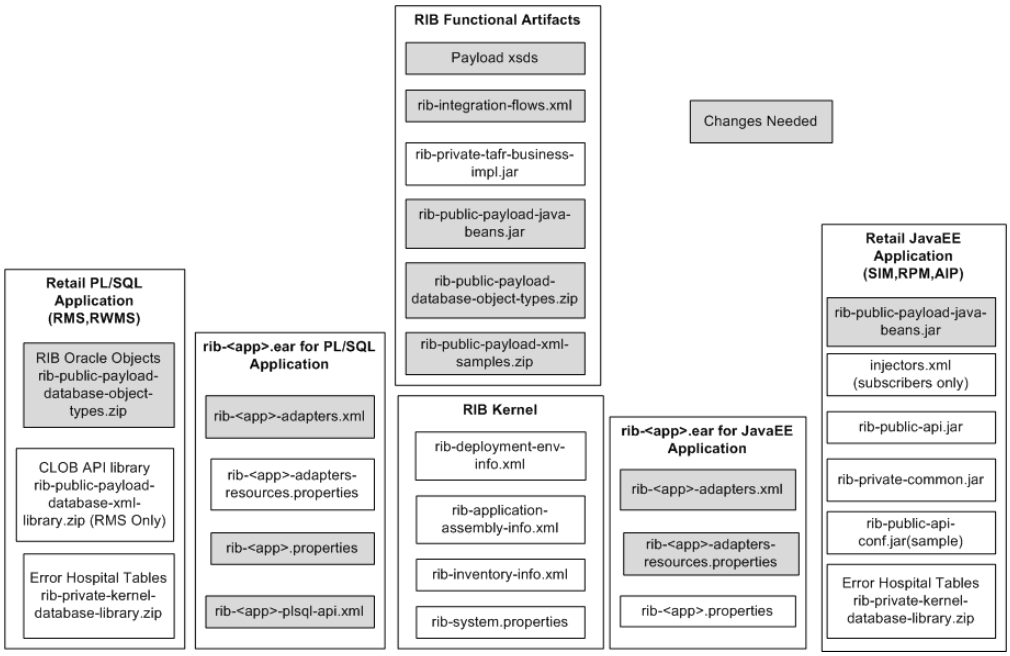
Creating a New Message Family

In the RIB, all messages are categorized by message family and message type. One option for customizing the RIB is to create a new message family with a new publishing adapter and a new subscribing adapter.

When creating a new message family, consider the following:

- If the RIB a family also corresponds to a topic, it is recommended that the customization also include the creation of a new topic for that family.
- A publishing adapter cannot publish to more than one JMS topic.
- A subscribing adapter cannot subscribe to more than one JMS topic.
- The first custom message flow must start with 901, with each subsequent custom messgae flow ID increasing by one from 901. For example, 901, 902, 903, and so on.
- Each customized message flow ID should be unique and must follow the sequence.
- A new message family requires new (or custom) Orace Retail application side APIs. Each API should be written, installed and tested independently, and then connected to the custom message family flows.

The following illustration indicates the files that require changes during the addition of new message family inside the RIB infrastructure.



Procedure for Adding a New Message Family

To add a new message family, complete the following steps.

1. Create a temporary working directory, "customization-workarea," under <RIB_HOME>/tools-home to perform any customization related tasks.
2. Copy the rib-func-artifact.war in <RIB_HOME>/application-assembly-home/rib-func-artifacts/ directory into <RIB_HOME>/tools-home/ customization-workarea/ directory.

```
cd <RIB_HOME>/application-assembly-home/rib-func-artifacts
cp rib-func-artifact.war <RIB_HOME>/tools-home/customization-workarea
```

3. Extract the rib-integration-flows.xml from the copied rib-func-artifact.war which needs to be modified.

```
cd <RIB_HOME>/tools-home/customization-workarea
jar -xvf rib-func-artifact.war integration/rib-integration-flows.xml
```

4. Define the entire flow for the particular message family in rib-integration-flows.xml in /integration/ directory of <RIB_HOME>/tools-home/ customization-workarea.

   A new custom message flow should always begin with <message-flow id="901">. Each customized message flow ID should be unique and must follow the sequence. Adding a new customized message flow with a message flow ID between 1 and 900 is not recommended, as this range is reserved for adding base flows in higher versions of RIB.

   For example, when adding a new message family, "Foo," that flows from RMS to RWMS, the flow is defined in rib-integration-flows.xml as follows:

```
<message-flow id="901">
            <node id="rib-rms.Foo_pub" app-name="rib-rms"
                adapter-class-def="Foo_pub" type="DbToJms">
                   <in-db>default</in-db>
                   <out-topic>etFooFromRMS</out-topic>
            </node>
         <node id=" rib-rwms.Foo_sub" app-name=" rib-rwms"
                  adapter-class-def="Foo_sub" type="JmsToDb">
                <in-topic>etFooFromRMS</in-topic>
                   <out-db>default</out-db>
            </node>
   </message-flow>
```

   The convention is as follows:

   - node id = rib-<app>.<family>_pub or = rib-<app>.<family>_sub or could be external-system.<family>_pub or external-system.<family>_sub
   - app-name = rib-<app>. This is the application name. The <app> value is one of the following: rms, rwms, sim, aip, rpm, tafr, or external-system
   - adapter-class-def = <family>_pub or <family>_sub
   - type = DbToJms or JmsToDb
   - <in-db> is the source of the message is a database.
   - <out-db> is the destination of the message is a database
   - <out-topic> is the topic name to which the message would be published
   - <in-topic> is the topic name from which the message would be consumed.

5. Replace the previous existing rib-integration-flows.xml with the above changed rib-integration-flows.xml to the /integration/ directory of rib-func-artifact.war in the <RIB_HOME>/tools-home/customization-workarea/ directory and generate the rib-func-artifact.war as follows.

```
cd <RIB_HOME>/tools-home/customization-workarea
jar -uvf rib-func-artifact.war integration/rib-integration-flows.xml
```

6. Create a new publishing adapter, subscribing adapter and TAFR adapter (if necessary), depending on the requirement for the new message family.

   Note: See "Adding New Adapters."

7. Create the message famly XSD.

Note: See "Adding a New Payload."

The newly created XSD should conform to the Meta schema, IntegrationXmlMetaSchema.xsd. The artifact generator tools check the validity of the schema before generating any artifacts. If the schema is not compliant with the IntegrationXmlMetaSchema, the artifact generator fails. The file is in the conf directory of the Artifact Generator tool installation program.

8. Create a new message type.

   Note: See "Adding a New Message Type."

9. Edit the payload.properties file in /conf directory of the Artifact Generator tool installation program. The payload.properties contains the new payload message definitions. The format of the definition is

   ```
   RIBFAMILY.TYPE=IMPLEMENTATION CLASS NAME
   cd conf
   vi payload.properties (make changes)
   ```

   For example, when adding a new message type, "FooCre," for the Foo message family that calls the implementation class, FooDesc, the payload.properties file is modified as follows:

   ```
   FOO.FOOCRE= com.retek.rib.binding.payload.FooDesc
   ```

10. Run the Artifact Generator to generate various functional artifacts

    ```
    $GROOVY_HOME/bin/groovy GenArtifacts.groovy
    ```

    Each generated artifact is in the appropriate ./output*/dist folder, including:

    - rib-public-payload-database-object-types.zip
    - rib-public-payload-java-beans.jar
    - retail-public-bo-java-beans.jar
    - rib-public-payload-xml-samples.zip

11. Copy the newly generated artifacts (listed above) from the appropriate ./output*/dist folders to the <RIB_HOME>/application-assembly-home/rib-func-artifacts/ directory.

12. New entries may be needed in the RIB_SETTINGS of the RMS application database to reference the new Message Family, only if the RMS application is in scope.

13. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows to create the new topic (etFooFromRMS) in our flow. (The step to prepare JMS is not destructive, so even if it is run again it would remove all the topics and recreate them.)

    ```
    cd <RIB_HOME>/deployment-home/bin
    sh rib-app-deployer.sh -prepare-jms
    ```

14. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

```
cd <RIB_HOME>/application-assembly-home/bin
sh rib-app-compiler.sh
```

15. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

```
cd <RIB_HOME>/deployment-home/bin
sh rib-app-deployer.sh -deploy-rib-func-artifact-war
```

This deploys the rib-func-artifact.war.

```
sh rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
```

The rib-<app> is deployed. Repeat this step for each rib-<app> that is in scope for this integration environment.

Note: The <app> value must be rms, rwms, tafr, sim, aip, or rpm.

Note: To verify the addition of a new message family, see "Verifying the New Message Family."

Adding New Adapters

A RIB Adapter is a component that coordinates business event (message) generation and processing with the respective Oracle Retail application interface. Each adapter in the RIB is created to handle a specific functional interface.

Note: See "Adapters" in Chapter 3 of the *Oracle Retail Integration Bus Operations Guide*.

Adding the Custom Adapter to the rib-integration-flows.xml File

When adding a custom publishing, subscribing or TAFR adapter, it is necessary to add or modify the message flows to which you are adding a custom adapter in the rib-integration-flows.xml. You also must update the rib-func-artifact.war and deploy the updated rib-func-artifact.war.

For example, when adding a new publisher, "Foo_pub," which publishes a message for the message family, Foo, that flows from RMS to RWMS, the flow in rib-integration-flows.xml is defined as follows:

```
<message-flow id="901">
            <node id="rib-rms.Foo_pub" app-name="rib-rms"
                adapter-class-def="Foo_pub" type="DbToJms">
                   <in-db>default</in-db>
                   <out-topic>etFooFromRMS</out-topic>
            </node>
          <node id=" rib-rwms.Foo_sub" app-name=" rib-rwms"
                    adapter-class-def="Foo_sub" type="JmsToDb">
                <in-topic>etFooFromRMS</in-topic>
                   <out-db>default</out-db>
            </node>
   </message-flow>
```

Procedure for Adding the Flow to the rib-integration-flows.xml File

> **Note:** Before adding the flow above to the rib-integration-flows.xml flow, it is recommended that a temporary working directory ("customization-workarea" under <RIB_HOME>/tools-home) be created. This directory can be used for performing any customization related tasks.

To add the flow to the rib-integration-flows.xml file, complete the following steps:

1. Copy the rib-func-artifact.war from the <RIB_HOME>/application-assembly-home/rib-func-artifacts to <RIB_HOME>/tools-home/customization-workarea/ directory.

   ```
   cd <RIB_HOME>/application-assembly-home/rib-func-artifacts
   cp rib-func-artifact.war  <RIB_HOME>/tools-home/customization-workarea
   ```

2. Extract the rib-integration-flows.xml requiring modification from the copied rib-func-artifact.war as follows:

   ```
   jar -xvf rib-func-artifact.war integration/rib-integration-flows.xml
   ```

3. Add the flow above to the rib-integration-flows.xml.

4. Update the rib-func-artifact.war with the modified rib-integration-flows.xml.

   ```
   jar -uvf rib-func-artifact.war integration/rib-integration-flows.xml
   ```

5. Copy the rib-func-artifact.war from the <RIB_HOME>/tools-home/customization-workarea to <RIB_HOME>/application-assembly-home/rib-func-artifacts/ directory.

   ```
   cd <RIB_HOME>/tools-home/customization-workarea
   cp rib-func-artifact.war  <RIB_
   HOME>/application-assembly-home/rib-func-artifacts
   ```

6. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

   ```
   cd <RIB_HOME>/application-assembly-home/bin
   sh rib-app-compiler.sh
   ```

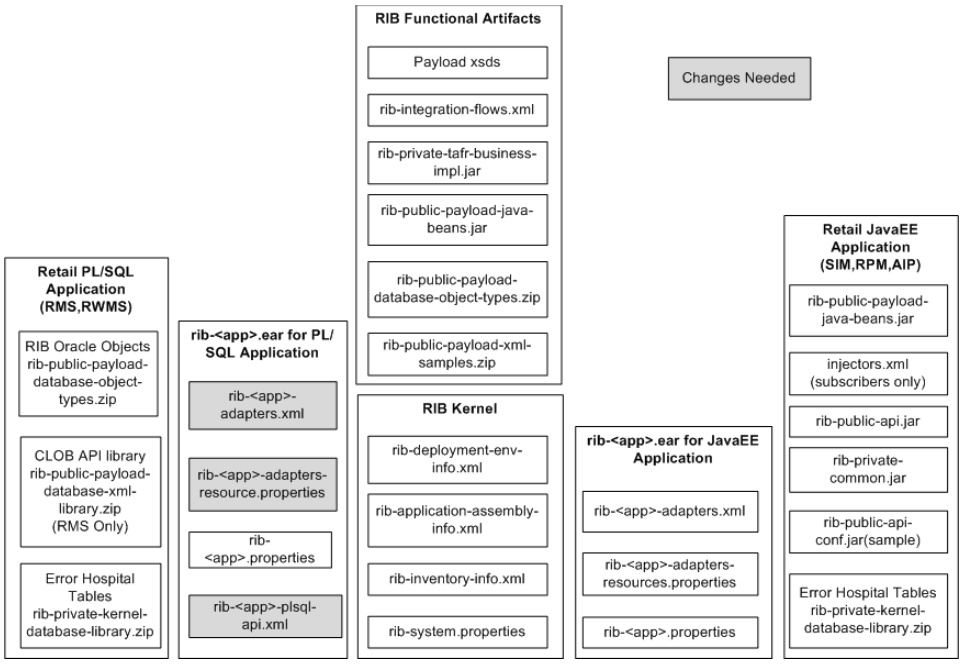
7. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

   ```
   cd <RIB_HOME>/deployment-home/bin
   sh rib-app-deployer.sh -deploy-rib-func-artifact-war
   ```

The rib-func-artifact.war is deployed.

Adding a Publishing Adapter for PL/SQL Applications

The illustraton below indicates the files that require changes inside the RIB infrastructure for the addition of a new publishing adapter for a PL/SQL application.



Procedure for Adding a Publishing Adapter for PL/SQL Applications

1. Identify the flow to which the new adapter in being added.
2. Define the name of the publishing adapter. It should always follow the naming convention, RIBFAMILY_pub_ADAPTER INSTANCE NO.
3. Define the particular publishing adapter in rib-<app>-adapters.xml in <RIB_HOME>/application-assembly-home/rib-<app>, where <app> refers to either RMS or RWMS. The customer also must mention a custom attribute equal to "true" whenever a new customized publishing adapter is added.

   For example, a new publishing adapter, "Foo_pub_1," for the Foo message family is defined in rib-<app>-adapters.xml as follows:

   ```
   <timer-driven id="Foo_pub_1" initialState="running" timeDelay="10"
   custom="true">
       <timer-task>
              <class name="com.retek.rib.app.getnext.impl.GetNextTimerTaskImpl"/>
              <property name="maxChannelNumber" value="1" />
       </timer-task>
   </timer-driven>
   ```

4. Define the particular publishing adapter in rib-<app>-adapters-resources.properties in <RIB_HOME>/application-assembly-home/rib-<app>, where <app> refers to either RMS or RWMS.

   ```
   Foo_pub_1.name=Foo Publisher, channel 1
   Foo_pub_1.desc=Publisher for the Foo family through channel 1.
   ```

5. Define the particular publishing adapter in rib-<app>-plsql-api.xml in <RIB_HOME>/application-assembly-home/rib-<app> where <app> refers to either RMS or RWMS as in below example.

Note: The signature of the stored procedure should come from the corresponding PL/SQL applications.

```
<adaptorClassDef name="Foo_pub">
 <class>com.retek.rib.collab.general.OracleObjectPublisherComponentImpl</class>
     <messageFamily name="Foo">
        <storedProc>
        <signature>{call RMSMFM_FOO.GETNXT(?,?,?,?,?,?,?,?)}</signature>
        </storedProc>
   </messageFamily>
</adaptorClassDef>
```

6. Make the required changes to the rib-integration-flows.xml. See Adding the Custom Adapter to the rib-integration-flows.xml File.
7. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

   ```
   cd <RIB_HOME>/application-assembly-home/bin
   sh rib-app-compiler.sh
   ```

8. Run the rib-app-builder deployer - Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

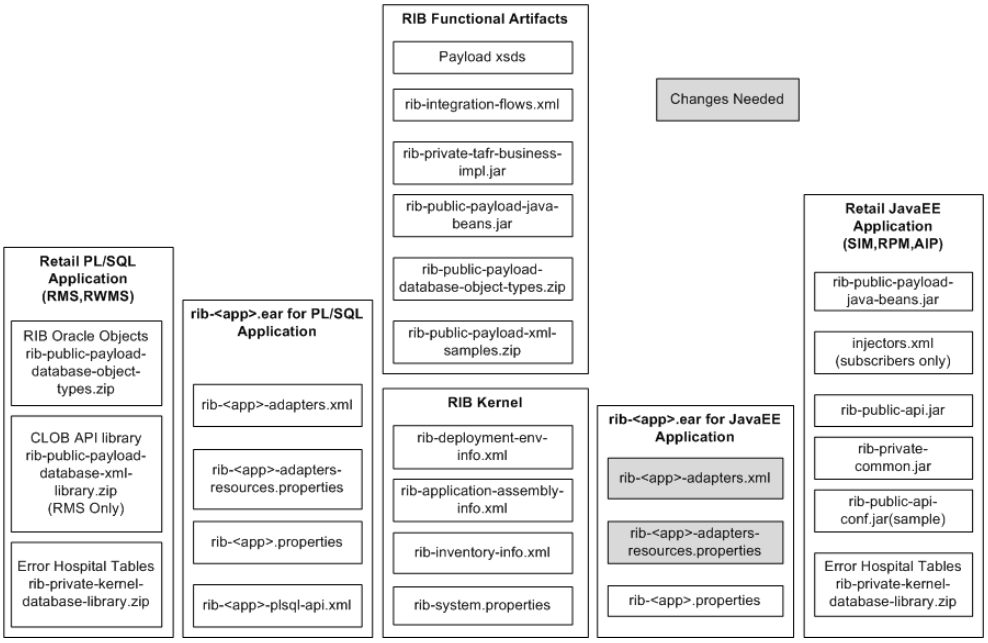
   ```
   cd <RIB_HOME>/deployment-home/bin
   sh rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
   ```

The <app> is deployed.

Note: To verify the addition of the new adapter, see "Verifying the New Publishing Adapter."

Adding a Publishing Adapter for Java EE Applications

The following illustration indicates the files that require changes inside the RIB infrastructure during the addition of a new publishing adapter for a Java EE application.



Procedure for Adding a Publishing Adapter for Java EE Applications

1. Identify the flow to which the new adapter in being added.
2. Define the name of the publishing adapter. It should always follow the naming convention, RIBFAMILY_pub_ADAPTER INSTANCE NO.
3. Define the particular publishing adapter in rib-<app>-adapters.xml in <RIB_HOME>/application-assembly-home/rib-<app>, where <app> refers to RPM, SIM, or AIP. The customer also must mention a custom attribute equal to "true" whenever a new customized publishing adapter is added.

   For example, a new publishing adapter, "Foo_pub_1," for the Foo message family is defined in rib-<app>-adapters.xml as follows.

   ```
   <request-driven id=" Foo_pub_1" initialState="notConfigurable" custom="true" />
   ```

4. Define the particular publishing adapter in rib-<app>-adapters-resources.properties in <RIB_HOME>/application-assembly-home/rib-<app>, where <app> refers to RPM, SIM, or AIP.

   ```
   Foo_pub_1.name=Foo Publisher, channel 1
   Foo_pub_1.desc=Publisher for the  Foo family through channel 1.
   ```

5. Make the required changes to the rib-integration-flows.xml. See Adding the Custom Adapter to the rib-integration-flows.xml File.

6. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

```
cd <RIB_HOME>/application-assembly-home/bin
sh rib-app-compiler.sh
```

7. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

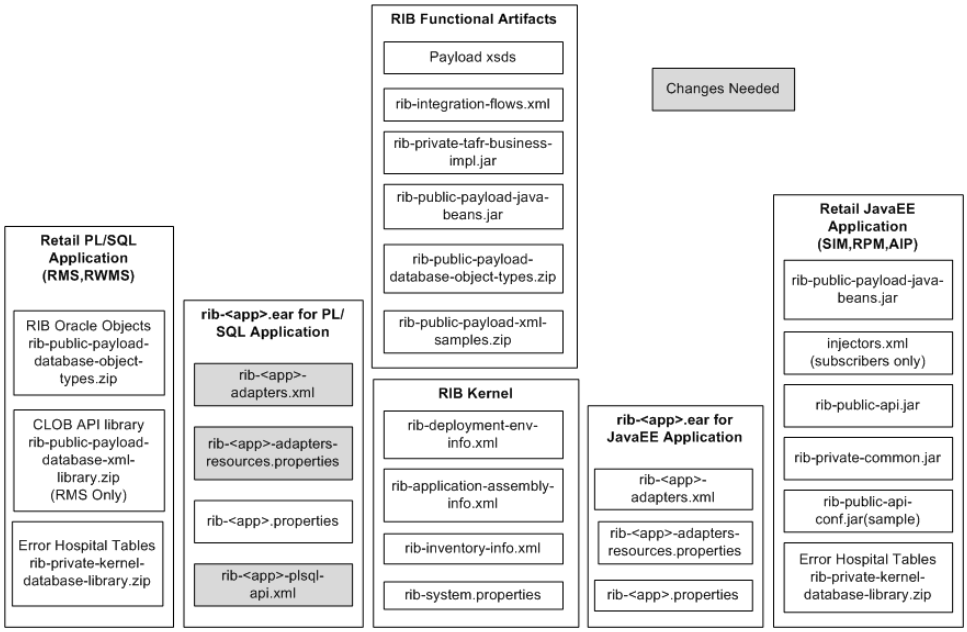
```
cd <RIB_HOME>/deployment-home/bin
sh rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
```

The <app> is deployed.

Note: To verify the addition of the new adapter, see "Verifying the New Publishing Adapter."

Adding a Subscribing Adapter for PL/SQL Applications

The following illustration indicates the files that require changes inside the RIB infrastructure during the addition of a new subscribing adapter for PL/SQL applications.



Procedure for Adding a New Subscribing Adapter for a PL/SQL Application

To add a new subscribing adapter for a PL/SQL application, complete the following steps:

1. Identify the flow to which the new adapter in being added.
2. Define the name of the subscribing adapter. It should always follow the naming convention, RIBFAMILY_sub_ADAPTER INSTANCE NO.

3. Define the particular subscribing adapter in rib-<app>-adapters.xml in <RIB_HOME>/application-assembly-home/rib-<app>, where <app> refers to either RMS or RWMS. The customer also must mention a custom attribute equal to "true" whenever a new customized subscribing adapter is added.

   For example, a new subscribing adapter, "Foo_sub_1," for the Foo message family, is defined in rib-<app>-adapters.xml as follows:

   ```
   <message-driven id="Foo_sub_1" initialState="running" custom="true"/>
   ```

4. Define the particular subscribing adapter in rib-<app>-adapters-resources.properties in <RIB_HOME>/application-assembly-home/rib-<app>, where <app> refers to either RMS or RWMS.

   ```
   Foo_sub_1.name= Foo Subscriber, channel 1
   Foo_sub_1.desc=Subscriber for the Foo family through channel 1.
   ```

5. Define the particular subscribing adapter in rib-<app>-plsql-api.xml in <RIB_HOME>/application-assembly-home/rib-<app> where <app> refers to either RMS or RWMS as shown below.

   Note: The signature of the stored procedure should come from the corresponding PL/SQL applications.

   ```
   <adaptorClassDef name="Foo_sub">
   <class>com.retek.rib.collab.general.OracleObjectSubscriberComponentImpl</class>
       <messageFamily name="Foo">
   <storedProc>
           <signature>{callRMSSUB_FOO.CONSUME(?,?,?,?)}</signature>
         </storedProc>
         <messageType name=" FOOCRE">
           <oracleObject>RIB_FooDesc_REC</oracleObject>
         </messageType>
       <messageType name=" FooMOD">
           <oracleObject>RIB_FooDesc_REC</oracleObject>
         </messageType>
         <messageType name=" FooDEL">
           <oracleObject>RIB_FooRef_REC</oracleObject>
        </messageType>
        </messageFamily>
     </adaptorClassDef>
   ```

6. Make the necessary changes to the rib-integration-flows.xml. See Adding the Custom Adapter to the rib-integration-flows.xml File.

7. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from the <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

   ```
   > cd <RIB_HOME>/application-assembly-home/bin
   > sh rib-app-compiler.sh
   ```

8. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from the <RIB_HOME>/deployment-home/bin directory as follows.

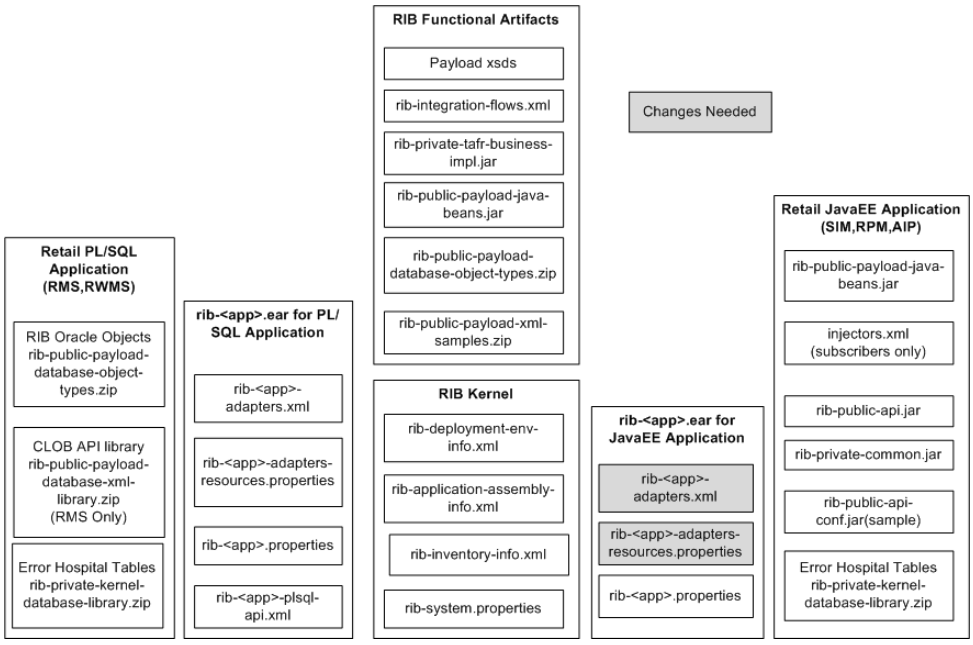
   ```
   > cd <RIB_HOME>/deployment-home/bin
   > sh rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
   ```

The <app> is deployed.

Note: To verify the addition of the new adapter, see "Verifying the New Subscribing Adapter."

Adding a Subscribing Adapter for Java EE Applications

The following illustration indicates the files that require changes inside the RIB infrastructure during the addition of a new subscribing adapter for a Java EE application.



Procedure for Adding a New Subscribing Adapter for a Java EE Application

To add a new subscribing adapter for a Java EE application, complete the following steps:

1. Identify the flow to which the new adapter in being added.
2. Define the name of the subscribing adapter. It should always follow the naming convention, RIBFAMILY_sub_ADAPTER INSTANCE NO.
3. Define the particular subscribing adapter in rib-<app>-adapters.xml in <RIB_HOME>/application-assembly-home/rib-<app>, where <app> refers to SIM or AIP. The customer also must mention a custom attribute equal to "true" whenever a new customized subscribing adapter is added.

   For example, a new subscribing adapter, "Foo_pub_1" for the Foo message family is defined in rib-<app>-adapters.xml as follows:

   ```
   <message-driven id="Foo_sub_1" initialState="running" custom="true" />
   ```

4. Define the particular subscribing adapter in rib-<app>-adapters-resources.properties in the <RIB_HOME>/application-assembly-home/rib-<app>, where <app> refers to SIM or AIP.

```
Foo_sub_1.name= Foo Subscriber, channel 1
Foo_sub_1.desc=Subscriber for the Foo family through channel 1.
```

5. Make the required changes to the rib-integration-flows.xml. See Adding the Custom Adapter to the rib-integration-flows.xml File.

6. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from the <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

```
> cd <RIB_HOME>/application-assembly-home/bin
> sh rib-app-compiler.sh
```

7. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

```
> cd <RIB_HOME>/deployment-home/bin
> sh rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
```

The <app> is deployed.

Note: To verify the addition of the new adapter, see "Verifying the New Subscribing Adapter."

Custom TAFR Adapters

Transformation Address Filters/Router (TAFR) adapters transform message data and route messages. Multiple, message family specific TAFRs have already been implemented. Different TAFR adapters may be active on different message families or on the same message family, depending on the needs of an application. Not all message families require TAFRs.

TAFR Considerations

The following topics should be considerd before writing a customized TAFR implementation for transformation, filtering or routing.

Transformation

Transformation is handled in the TAFR implementation class. The following is an example of a TAFR that handles transformation.

```
public RibMessage transformRibMessage(RibMessage inMsg) throws TafrException {
// Transforms the incoming RibMessage into an outgoing RibMessage
RibMessage newMsg = transform(inMsg);
return newMsg; }
```

Filtering Configuration

Filtering configuration involves updating the rib-tafr.properties file with the appropriate information. The property follows the usual properties naming convention (name=value). The property used for filtering is:

```
"for.<tafr name>_tafr.drop-messages-of-types"
```

Example:

```
for.ItemsToItemsISO_
tafr.drop-messages-of-types=ISCDimCre,ISCDimMod,ISCDimDel,ItemImageCre,ItemImageMo
d,ItemImageDel,ItemUdaDateCre,ItemUdaDateMod,ItemUdaDateDel,ItemUdaFfCre,ItemUdaFf
Mod,ItemUdaFfDel,ItemUdaLovCre,ItemUdaLovMod,ItemUdaLovDel
```

This property should be read as, "for ItemsToItemsISO tafr, drop these message types." A comma delimits the message types. If customization is required, rib-tafr.properties files must be updated for filtering to take place.

Routing

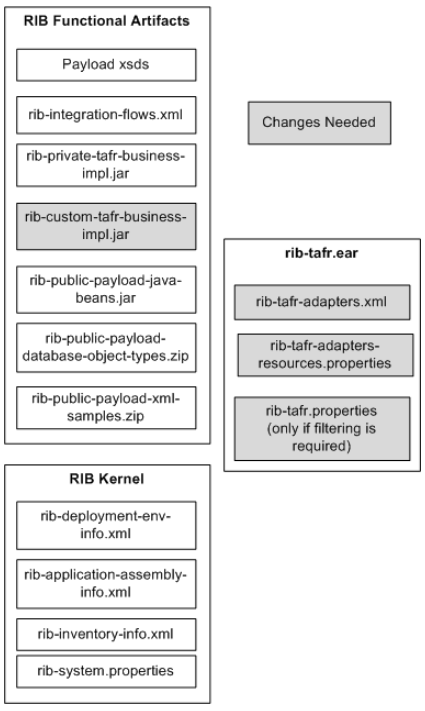
Routing is enabled by default for TAFRs; the RIB infrastructure handles this routing. If a TAFR requires routing based on message content, implementation classes override the following method.

```
public void routeRibMessage(RibMessage newMsg,MessageRouterIface router) throws
TafrException {
    router.addMessageForTopic(eventType, newMsg);
}
```

Adding a New TAFR Adapter

This section explains how to create a new TAFR adapter for a particular message family.

The following illustration indicates the files that require changes inside the RIB infrastructure during the addition of new TAFR adapter to a message family.



Procedure for Adding a New TAFR Adapter

To add a new TAFR adapter, complete the following steps.

1. Identify the flow to which the new adapter in being added.
2. Define the name of the TAFR adapter. It should always follow the naming convention, RIBFAMILY_tafr_ADAPTER INSTANCE NO.
3. Define the corresponding implementation class name the TAFR needs to call.
4. Write the implementation class for the TAFR.

Custom TAFR Implementation

The default implementation of a TAFR implements the following interface in the RIB infrastructure.

```
package com.retek.rib.collab.tafr;

import com.retek.rib.domain.ribmessage.bo.RibMessage;

public interface TafrIface {
@return ribMessage that has been modified from the original one
public RibMessage transformRibMessage(RibMessage ribMsgIn) throws TafrException;

/**
 * Filters message or messages contents accordingly. It is possible that
 * this method could filter away the entire message thus returning null
 * from this method.
 *
 * @param ribMsg
 * @return ribMessage that may have been modified from the original one
 *  passed in or null.
 */
public RibMessage filterRibMessage(RibMessage ribMsgIn) throws TafrException;

/**
 * Routes the message to the appropriate topic for publication.
 *
 * @param ribMsg RibMessage to be routed to the appropriate topic.
 */
public void routeRibMessage(RibMessage ribMsgIn, MessageRouterIface
router) throws TafrException;

public void processRibMessage(RibMessage ribMsgIn, MessageRouterIface
router) throws TafrException;
}
```

Procedure for Completing Custom TAFR Implementation

To implement a custom TAFR, complete the following steps:

1. Verify that the default implementation that comes with the RIB infrastructure is appropriate.
2. Create a rib-custom-tafr-business-impl.jar containing the customized implementation class for the specific message family and replace the same under <RIB_HOME>/application-assembly-home/rib-func-artifacts.

Note: See Metalink Note 837997.1, "How to Create a Custom TAFR Implementation."

3. Define the particular TAFR adapter in rib-tafr-adapters.xml in <RIB_HOME>/application-assembly-home/rib-tafr. The customer must mention a custom attribute equal to "true" whenever a new customized TAFR adapter is added.

   For example, when adding a new TAFR adapter, "Foo_tafr_1," for the Foo message family, the implementation class written is "SampleToSampleWH." The class is in com.retek.rib.collab.tafr.bo.impl, inside rib-custom-tafr-business-impl.jar. It should be defined in rib-tafr-adapters.xml as follows:

   ```
   <message-driven id="Foo_tafr_1" initialState="running" tafr-business-
   impl="com.retek.rib.collab.tafr.bo.impl.SampleToSampleWH"
   custom="true" />
   ```

4. Define the particular TAFR adapter in rib-tafr-adapters-resources.properties in the <RIB_HOME>/application-assembly-home/rib-tafr.

   ```
   Foo_tafr_1.name=Foo TAFR, channel 1
   Foo_tafr_1.desc=TAFR for the Foo family through channel 1.
   ```

5. Make the required changes to the rib-integration-flows.xml. See "Adding the Custom Adapter to the rib-integration-flows.xml File."

6. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from the <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

   ```
   cd <RIB_HOME>/application-assembly-home/bin
   sh rib-app-compiler.sh
   ```

7. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

   ```
   cd <RIB_HOME>/deployment-home/bin
   sh rib-app-deployer.sh -deploy-rib-app-ear rib-tafr
   ```

Note: To verify the addition of the new TAFR adapter, see "Verifying the New TAFR Adapter."

Changing an Existing TAFR Adapter

If there is a need to add more functionality to an existing TAFR than what is already provided, a class can be added to extend from the original TAFR class.

Procedure for Changing an Existing TAFR Adapter

To change an existing TAFR adapter, complete the following steps:

1. Identify the TAFR to which more functionality should be added.
2. Define the corresponding implementation class name the TAFR needs to call. This class should extend from the original TAFR implementation class.

For example:

- Additional functionality has to be added to the ASNOutToASNIn_tafr_1 TAFR with an implementation class of ASNOutToASNInLocFromRibBOImpl.
- A new class should be written for the additional functionality that extends from ASNOutToASNInLocFromRibBOImpl.
- If additional functionality is needed for the transformation of the message, call the transform method of the ASNOutToASNInLocFromRibBOImpl class and write your own code/logic.

Note: For information on how to write the implementation class., see the Metalink Note 837997.1, "How to Create a Custom TAFR Implementation."

3. Write the implementation class for the TAFR.
4. Create a rib-custom-tafr-business-impl.jar containing the implementation class and replace the same under <RIB_HOME>/application-assembly-home/rib-func-artifacts.

Note: For more information on how to create the rib-custom-tafr-business-impl.jar, see the Metalink Note 837997.1, "How to Create a Custom TAFR Implementation."

5. Replace the name of the implementation class with the new class name in the rib-tafr-adapters.xml as shown below. For example, if the name of the new class name is CustomASNOutToASNInLocFromRibBOImpl, the entry in tafr-adapters.xml should be:

```
<message-driven id="ASNOutToASNIn_tafr_1" initialState="running"
tafr-business-impl=" com.retek.rib.collab.tafr.bo.impl.
CustomASNOutToASNInLocFromRibBOImpl "  custom ="true"/>
```

6. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from the <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

```
cd <RIB_HOME>/application-assembly-home/bin
sh rib-app-compiler.sh
```

7. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

```
cd <RIB_HOME>/deployment-home/bin
sh rib-app-deployer.sh -deploy-rib-app-ear rib-tafr
```

Verification of RIB Customizations

This section explains how to verify the various customizations using the RIB diagnostic and test tools, RDMT, the PL/SQL API simulator, and the Java EE API simulator.

These verification tests are described only from a RIB perspective and not as end-to-end testing. They should be considered only the first step in a process to move the customizations through the RIB life cycle.

The verification steps assume that these RIB tools have already been installed and are in working condition.

Note: See "Testing the RIB" in the *Oracle Retail Integration Bus Operations Guide.*

Verifying the New Message Type

To verify the addition of a new message type under a message family from a RIB perspective, complete the following steps:

1. Log in to the RDMT main menu.
2. Select menu option 3, PUB/SUB/TAFR Utilities Submenu.
3. Publish a message using option 8, EJB Publish Utility.
4. Provide the new message type when prompted for the <type> parameter.
5. Use the sample message that was generated using the Artifact Generator tool after adding the new message type for the corresponding message family.

   Check the corresponding adapter's RIBLOGS to be sure the message was published successfully. The logs are written to the path, <rib-application_instance_home>/<rib-app>/logs/<rib-app>.

   For example, for /home/rib/product/10.1.3.3/OracleAS_1/j2ee/rib-rms-oc4j-instance/log/rib-rms, the RIBLOG file names are in the format, <adapter-instance-name>.rib.log.

   Example:

   ```
   Alloc_pub_1.rib.log
   ASNIn_sub_1.rib.log
   ```

6. Enable the RIB Audit Logs for all the corresponding adapters involved in the message flow. The auditing feature logs the message as it passes through the RIB infrastructure. This helps the tracing of message content from publication to subscription, and all steps, such as a TAFR, in between.

   Note: To enable RIB Audit logs, see "RIB Logging" in the *Oracle Retail Integration Bus Operations Guide*.

7. Check the RIB audit logs for that particular message family adapter (publisher, subscriber, and TAFR, if involved) and verify whether the new message type added is part of the message header. Also ensure that the message passes successfully through all the adapters involved in the particular message flow.
8. Check whether the new message type was successfully consumed by the subscribing adapter.

Verifying the New Message Family

To verify the addition of a new message family in the RIB, complete the following steps.

1. Once the RIB has been compiled and deployed (after adding new message family), check whether the new family adapters (publisher, subscriber, and TAFR, if involved) are visible through RIB Admin GUI.

The RIB admin GUI can be accessed via the URL as below.

```
http://<server>.us.oracle.com:<http-port>/rib-<app>-admin-gui/
```

- Replace <server> with the name or IP address of the server in the environment where the rib-<app> deployed.
- Replace <http-port> with the port number that the Oracle Application Server is listening on (for example, 7777).
- Replace <app> with rms, tafr, rwms, sim, rpm, or aip.

Note: See "Admin GUI" in the *Oracle Retail Integration Bus Operations Guide*.

2. Log in to the RDMT main menu.
3. Select menu option 3, PUB/SUB/TAFR Utilities Submenu.
4. Publish a message using option 8, EJB Publish Utility.
5. Provide the new message family when prompted for the <family> parameter.
6. Use the sample message created by the Artifact Generator tool for the corresponding new message family.
7. Check the corresponding adapter's RIBLOGS to be sure the message was published successfully. The logs are written to the path, <rib-application_instance_home>/<rib-app>/logs/<rib-app>.

   For example, for /home/rib/product/10.1.3.3/OracleAS_1/j2ee/rib-rms-oc4j-instance/log/rib-rms, the RIBLOG file names are in the format, <adapter-instance-name>.rib.log.

   Example:

   ```
   Foo_pub_1.rib.log
   Foo_sub_1.rib.log
   ```

8. Enable the RIB Audit Logs for all the corresponding adapters involved in the message flow. The auditing feature logs the message as it passes though the RIB infrastructure. This helps the tracing of message content from publication to subscription, and all steps, such as a TAFR, in between.

   Note: To enable RIB Audit logs, see "RIB Logging" in the *Oracle Retail Integration Bus Operations Guide*.

9. Check the RIB audit logs for the particular message family adapters (publisher, subscriber and TAFR, if involved) and verify whether the new message family added is part of the message header. Also ensure that the message passes successfully through all the adapters involved in the particular message flow.

Verifying the New Publishing Adapter

To verify the addition of a new publishing adapter for PL/SQL for Java EE applications, complete the following steps.

1. Once the RIB has been compiled and deployed (after adding new publshing adapter), check whether the new publishing adapter is visible through RIB Admin GUI.

   The RIB admin GUI can be accessed through the URL below.

   ```
   http://<server>.us.oracle.com:<http-port>/rib-<app>-admin-gui/
   ```

   - Replace <server> with the name or IP address of the server in the environment where the rib-<app> is deployed.
   - Replace <http-port> with the port number that the Oracle Application Server is listening on (for example, 7777).
   - Replace <app> with rms, rwms, sim, rpm, or aip.

   Note: See "Admin GUI" in the *Oracle Retail Integration Bus Operations Guide.*

2. Log in to the RDMT main menu.
3. Select menu option 3, PUB/SUB/TAFR Utilities Submenu.
4. Publish a message using option 8, EJB Publish Utility.
5. Use the sample message created by the Artifact Generator tool for the corresponding message family.
6. Check the corresponding publishing adapter's RIBLOGS to be sure the message was published successfully. The logs are written to the path, <rib-application_instance_home>/<rib-app>/logs/<rib-app>.

   For example, for /home/rib/product/10.1.3.3/OracleAS_1/j2ee/rib-rms-oc4j-instance/log/rib-rms, the RIBLOG file names are in the format, <adapter-instance-name>.rib.log.

   Example:

   ```
   Foo_pub_1.rib.log
   ```

7. Enable the RIB Audit Logs for the corresponding publishing adapter involved in the message flow. The auditing feature logs the message as it passes though the RIB infrastructure. This helps the tracing of message content from publication to subscription.

   Note: To enable RIB Audit logs, see the section "RIB Logging" in the *Oracle Retail Integration Bus Operations Guide*.

8. Check the RIB audit logs for that particular message family's publishing adapter and verify whether the message content is displayed correctly as published. Also ensure that the message passes successfully through all the adapters involved in the particular message flow.

Verifying the New Subscribing Adapter

To verify the addition of a new subscribing TAFR adapter for PL/SQL and Java EE applications, complete the following steps:

1. Once the RIB has been compiled and deployed after adding new subscribing adapter, check whether the new subscribing adapter is visible through RIB Admin GUI.

   The RIB admin GUI can be accessed through the URL below.

   ```
   http://<server>.us.oracle.com:<http-port>/rib-<app>-admin-gui/
   ```

   - Replace <server> with the name or IP address of the server in the environment where the rib-<app> is deployed.
   - Replace <http-port> with the port number that the Oracle Application Server is listening on (for example, 7777).
   - Replace <app> with rms, rwms, sim, rpm, or aip.

   Note: See "Admin GUI" in the *Oracle Retail Integration Bus Operations Guide*.

2. Log in to the RDMT main menu.
3. Select menu option 3, PUB/SUB/TAFR Utilities Submenu.
4. Publish a message using option 1, Publish Msg Utility, to the topic from which the newly added subscriber has to subscribe.
5. Use the sample message created by the Artifact Generator tool for the corresponding message family.
6. Check the corresponding subscribing adapter's RIBLOGS to check if the message was subscribed from the topic successfully. The logs are written to the path, <rib-application_instance_home>/<rib-app>/logs/<rib-app>.

   For example, for /home/rib/product/10.1.3.3/OracleAS_1/j2ee/rib-rms-oc4j-instance/log/rib-rms, the RIBLOG file names are in the format, <adapter-instance-name>.rib.log.

   Example:

   ```
   Foo_sub_1.rib.log
   ```

7. Enable the RIB Audit Logs for the corresponding subscribing adapter. The auditing feature logs the message as it passes though the RIB infrastructure. This helps the tracing of message content from publication to subscription.

   Note: To enable RIB Audit logs, see "RIB Logging" in the *Oracle Retail Integration Bus Operations Guide.*

8. Check the RIB audit logs for the particular message family's subscribing adapter and verify whether the message content is displayed correctly. Also ensure that the message is subscribed successfully by the subscribing adapter

Verifying the New TAFR Adapter

To verify the addition of a new TAFR adapter, complete the following steps:

1. Once the RIB has been compiled and deployed after adding new TAFR adapter, check whether the new TAFR adapter is visible through RIB Admin GUI.

   The RIB admin GUI can be accessed via the URL as below.

   ```
   http://<server>.us.oracle.com:<http-port>/rib-tafr-admin-gui/
   ```

   - Replace <server> with the name or IP address of the server in the environment that has the rib-<app> deployed.
   - Replace <http-port> with the port number that the Oracle Application Server is listening on (for example, 7777).

   Note: See "Admin GUI" in the *Oracle Retail Integration Bus Operations Guide*.

2. Log in to the RDMT main menu.
3. Select menu option 3, PUB/SUB/TAFR Utilities Submenu.
4. Publish a message using option 1, Publish Msg Utility, to the topic from which the newly added subscriber has to subscribe.
5. Use the sample message created by the Artifact Generator tool for the corresponding message family.
6. Check the corresponding TAFR adapter's RIBLOGS to be sure the message was subscribed by the TAFR from the particular topic and published to the next destination topic successfully. The logs are written to the path, <rib-application_instance_home>/rib-tafr/logs/rib-tafr.

   For example, for/home/rib/product/10.1.3.3/OracleAS_1/j2ee/rib-tafr-oc4j-instance/log/rib-tafr, the RIBLOG file names are in the format, <adapter-instance-name>.rib.log.

   Example:

   ```
   Foo_tafr_1.rib.log
   ```

7. Enable the RIB Audit Logs for the corresponding TAFR adapter. The auditing feature logs the message as it passes though the RIB infrastructure. This helps the tracing of message content from publication to subscription.

   Note: To enable RIB Audit logs, see "RIB Logging" in the *Oracle Retail Integration Bus Operations Guide*.

8. Check the RIB audit logs for the particular message family's TAFR adapter and verify whether the message content is displayed correctly.

Payload Customization

The customization of payloads gives a customer the ability to add/modify data which flows from one application to the other.

> **Note:** The Artifact Generator tool must be installed before performing any payload customizations. See the *Oracle Retail Integration Bus Operations Guide* for information on the installation and usage of the Artifact Generator tool.

Prerequisites

Individuals performing the tasks for payload customization should be familiar with the Artifacts Generator tool and have an understanding of the following:

- The importance of payloads and how they fit into the suite of Oracle Retail applications.
- The impact that customizing a payload has on other applications.

Recommendations

The following is a list of recommendations for ensuring successful payload customization.

- Always back up the files that will be modified during customization in case they need to be restored.
- When payloads are customized, the changes applied must also be made on the application side.
- During customization, only optional elements should be added. The addition of mandatory elements to payloads can result in increased maintenance efforts.
- Names of elements in XSDs must not be Java key words.

Adding Optional Elements to Payloads

To add of an optional element (simple type or complex type) to an existing message payload, complete the following steps:

1. Edit the desired payload XSDs in the ./input-xsd directory of the Artifact Generator tool installation. Add the optional simple or complex element to the particular message family XSD and, if necessary, define the type.

   ```
   cd input-xsd
   vi ItemRef.xsd (make changes)
   ```

   For example, to add an optional simple element, "attr1," and a complex element, "ItemDtlRef," to ItemRef.xsd, the modification to ItemRef.xsd is as follows.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <xs:schema elementFormDefault="qualified"
       targetNamespace="http://www.oracle.com/retail/integration/payload/ItemRef"
       xmlns="http://www.oracle.com/retail/integration/payload/ItemRef"
       xmlns:retailDoc="http://www.w3.org/2001/XMLSchema"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="ItemDtlRef">
           <xs:complexType>
               <xs:sequence>
   ```

```
                <xs:element maxOccurs="unbounded" name="clearance_id"
type="number10">
                </xs:element>
                <xs:element maxOccurs="unbounded" name="item"
type="varchar225">
                </xs:element>
            </xs:sequence>
        </xs:complexType>
 </xs:element>
 <xs:element name="ItemRef">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="item" type="varchar225"/>
        <xs:element name="attr1" type="number10" minOccurs="0"/>
        <xs:element maxOccurs="unbounded" minOccurs="0" ref="ItemDtlRef"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="varchar225">
    <xs:restriction base="xs:string">
      <xs:maxLength value="25"/>
    </xs:restriction>
  </xs:simpleType>
 <xs:simpleType name="number10">
    <xs:restriction base="xs:long">
     <xs:totalDigits value="10"/>
    </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

2. Run the Artifact Generator to generate various functional artifacts.

   ```
   $GROOVY_HOME/bin/groovy GenArtifacts.groovy
   ```

   Each generated artifact is in the appropriate ./output*/dist folder, including:

   - rib-public-payload-database-object-types.zip
   - rib-public-payload-java-beans.jar
   - retail-public-bo-java-beans.jar
   - rib-public-payload-xml-samples.zip

   Note: See the *Oracle Retail Integration Bus Operations Guide* for information on how to use the Artifact Generator tool to generate artifacts.

3. Copy the newly generated artifacts (listed above) from the appropriate ./output*/dist folders to the <RIB_HOME>/application-assembly-home/rib-func-artifacts/ directory.

4. The artifact generator tool cannot be used to generate the rib-func-artifact.war. The file must be created manually as follows:

   a. Copy the rib-func-artifact.war from <RIB_HOME>/application-assembly-home/rib-func-artifacts to <RIB_HOME>/tools-home/customization-workarea/ directory.

      ```
      cd <RIB_HOME>/application-assembly-home/rib-func-artifacts
      cp rib-func-artifact.war  <RIB_HOME>/tools-home/customization-workarea
      ```

b. Extract only the XSD that must be modified from the copied rib-func-artifact.war, as shown below.

```
jar -xvf rib-func-artifact.war payload/xsd/ItemRef.xsd
```

c. Copy and replace the same modified payload XSD from ./input-xsd directory of the Artifact Generator tool installation program to the <RIB_HOME>/tools-home/customization-workarea/ payload/xsd directory .

```
cd ArtifactGeneratorInstallation_directory/input-xsd
cp ItemRef.xsd  <RIB_HOME>/tools-home/customization-workarea/payload/xsd
```

After replacing the existing ItemRef.xsd with the changed ItemRef.xsd to the /payload/xsd directory of rib-func-artifact.war in the <RIB_HOME>/tools-home/customization-workarea/ directory, generate the new rib-func-artifact.war as shown below.

```
cd  <RIB_HOME>/tools-home/customization-workarea
jar -uvf rib-func-artifact.war payload/xsd/ItemRef.xsd
```

d. Copy the generated rib-func-artifact.war from <RIB_HOME>/tools-home/customization-workarea to the <RIB_HOME>/application-assembly-home/rib-func-artifacts/ directory.

```
cd <RIB_HOME>/tools-home/customization-workarea
cp rib-func-artifact.war  <RIB_
HOME>/application-assembly-home/rib-func-artifacts
```

e. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from the <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

```
cd <RIB_HOME>/application-assembly-home/bin
sh rib-app-compiler.sh
```

Note: See the section, "rib-app-builder Tools," in the *Oracle Retail Integration Bus Operations Guide*.

---

5. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

```
cd <RIB_HOME>/deployment-home/bin
sh rib-app-deployer.sh -deploy-rib-func-artifact-war
```

This step deploys the rib-func-artifact.war to the Java EE container.

```
sh rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
```

The rib-<app> is deployed to the Java EE container. Repeat this step for each rib-<app> in scope for this integration environment.

Note: The <app> value must be rms, rwms, tafr, sim, aip, or rpm.

Adding a New Payload

The following steps must be completed to add a new XSD to a current set of payloads:

1. Create a new XSD which conforms to the MetaSchema ie IntegrationMetaSchema.xsd. The artifact generater tool checks the validity of the schema before generating any artifacts. The artifact generater will fail if the XSD is not compliant with the MetaSchema.
2. Drop the new XSD in here: ./input-xsd/ directory of the Artifact Generator tool installation.

   For example, if you create a XSD called "Foo.xsd", place it under ./input-xsd/.
3. Run the Artifact Generator to generate various functional artifacts.

   ```
   $GROOVY_HOME/bin/groovy GenArtifacts.groovy
   ```

   Each generated artifact is in the appropriate ./output*/dist folder, including:

   - rib-public-payload-database-object-types.zip
   - rib-public-payload-java-beans.jar
   - retail-public-bo-java-beans.jar
   - rib-public-payload-xml-samples.zip
4. Copy the newly generated artifacts (listed above) from the appropriate ./output*/dist folders to the <RIB_HOME>/application-assembly-home/rib-func-artifacts/ directory.

   Note: See the *Oracle Retail Integration Bus Operations Guide* for information on how to use the Artifact Generator tool to generate artifacts.
5. The artifact generator tool cannot be used to generate the rib-func-artifact.war. The file must be created manually as follows:

   a. Copy the rib-func-artifact.war from <RIB_HOME>/application-assembly-home/rib-func-artifacts to the <RIB_HOME>/tools-home/customization-workarea/ directory.

      ```
      cd <RIB_HOME>/application-assembly-home/rib-func-artifacts
      cp rib-func-artifact.war  <RIB_HOME>/tools-home/customization-workarea
      ```

   b. Copy the newly created XSD (Foo.xsd) to the /payload/xsd/ directory under <RIB_HOME>/tools-home/customization-workarea/ directory and update the rib-func-artifact.war.

      ```
      cd <RIB_HOME>/tools-home/customization-workarea
      jar -uvf rib-func-artifact.war payload/xsd/Foo.xsd
      ```

   c. Copy the generated rib-func-artifact.war from <RIB_HOME>/tools-home/customization-workarea to the <RIB_HOME>/application-assembly-home/rib-func-artifacts/ directory.

      ```
      cd <RIB_HOME>/tools-home/customization-workarea
      cp rib-func-artifact.war  <RIB_
      HOME>/application-assembly-home/rib-func-artifacts
      ```

6. Run the rib-app-builder compiler: Run the rib-app-compiler.sh script from <RIB_HOME>/application-assembly-home/bin directory to generate/assemble a rib-<app> and make it ready for deployment.

```
cd <RIB_HOME>/application-assembly-home/bin
sh rib-app-compiler.sh
```

7. Run the rib-app-builder deployer: Run the rib-app-deployer.sh script from <RIB_HOME>/deployment-home/bin directory as follows.

```
cd <RIB_HOME>/deployment-home/bin
sh rib-app-deployer.sh -deploy-rib-func-artifact-war
```

   This deploys the rib-func-artifact.war to the Java EE container.

```
sh rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
```

The rib-<app> is deployed to the Java EE container. Repeat this step for all rib-<app> that is in scope for this integration environment.

Note: The <app> value must be rms, rwms, tafr, sim, aip, or rpm.