Oracle® Retail Integration Cloud Service

Implementation Guide–Concepts

Release 24.0.101.0
F90417-03
May 2024

ORACLE

Oracle Retail Integration Cloud Service Implementation Guide–Concepts, Release 24.0.101.0

F90417-03

Contents

A Appendix - Sample Files

Send Us Your Comments

Oracle Retail Integration Cloud Service Implementation Guide–Concepts, Release 23.1.401.0

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

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

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

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

> **Note:**
>
> Before sending us your comments, you might like to check that you have the latest version of the document and if any concerns are already addressed. To do this, access the new Applications Release Online Documentation CD available on My Oracle Support and www.oracle.com. It contains the most current Documentation Library plus all documents revised or released recently.

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

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

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

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

Preface

The *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.

Documentation Accessibility

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

Access to Oracle Support

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

Customer Support

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

https://support.oracle.com

When contacting Customer Support, please provide the following:

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

Improved Process for Oracle Retail Documentation Corrections

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

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

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

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

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

Oracle Retail Documentation on the Oracle Help Center (docs.oracle.com)

Oracle Retail product documentation is also available on the following Web site:

https://docs.oracle.com/en/industries/retail/index.html

(Data Model documents can be obtained through My Oracle Support.)

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

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.

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.

The Integration Gateway Services (IGS) and RIB-ext components provides an integration infrastructure for external system (3rd Party) connectivity to the Oracle Retail Integration Bus (RIB) in the form of a tested set of Web service providers and the configurations to connect to RIB.

The issues and considerations needed to properly deploy and configure the integration solution within an enterprise are the subject of this guide.

2

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 as 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 Order 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 *Oracle Retail Integration Bus 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, among 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 *Oracle Retail Integration Bus Integration Guide*.

Functional Area	Publishing Applications	Subscribing Applications
Allocations	RMS	RWMS, SIM
Appointments	RWMS	RMS, SIM
ASN Outbound	RWMS, SIM, RMS, RFM	RMS, SIM, RWMS,
ASN Inbound	RWMS, External, RMS RFM	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, SIM
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.

> **Note:**
>
> Payloads do not support time zone formats.

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. Similarly, SOAP Applications calls with the payload information in the request 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, the subscribing application's data can be corrupted.

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 (because it has yet to be processed successfully), the adapter now recognizes this message is problematic and checks it into the hospital. If adding the message to error hospital fails, the subscribing adapter writes the message to the file system. This becomes a poison message (.xml).

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. Subscribing retry adapter also processes the poison message. It extracts the message from the poison-message file and adds it to the error hospital to be retried. The poison message file will be renamed to processed message (.processed). If the retry adapter fails to process the poison message, the file is moved to human-workflow file (.humanworkflow).

The unprocessed poison messages should be corrected with a human intervention. They are made available in object storage bucket at a regular interval. These messages should be downloaded from object-store, corrected and uploaded back to object store. RIB will process these uploaded messages through subscriber retry adapter.

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).

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-queue application is known as publisher, and the de-queue 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 other's 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 allow the Oracle Retail edge applications to be in up or down state independently but still not lose 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 Reports section of the *Oracle Retail Integration Bus Integration Guide*.

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.

Pseudo code for message selector:

```
(
      (
        (appName is not null) AND
        (appName == $APP_NAME)
      ) AND
      (
        (retryLocation is not null) AND
        (retryLocation LIKE $ADP_CLASS_DEF)
      )
    ) OR
    (
     (
```

```
    (appName is null) OR
    (appName != $APP_NAME)
  ) AND
  (
    (retryLocation is null) OR
    (retryLocation LIKE $ADP_CLASS_DEF)
  )
) AND
(threadValue == $ADP_INSTANCE_NUMBER)
```

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 name of the RIB family name to which the message belongs.

- 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 has 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, make our processing parallel, 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=22.0

  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=22.0

  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. 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 or a request in case of SOAP application. 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 that message 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 topics. There may be new subscribers to the modified message, and the scenario is 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.

Two types of applications require this data and subscribe to it. One type of subscribing application requires a certain transformation be applied to the data, but the other type of 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 is 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-publish 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 the Oracle Retail application documentation (for example, "Message Publication and Subscription Designs" in the *Oracle Retail Merchandising System Operations Guide Volume 2*).

RIB Hospital Insert

In an event of failure during message subscription, the error is flagged within the RIB Hospital software, resulting in rollback of the work done in the retail application, the adapter returns failure so that the database transaction is rolled back as well, and the message is kept on the integration bus topic. 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. Immediately after the roll back, JMS re-delivers the message back to the subscribing adapter and this time the RIB Hospital software detects the previously flagged message and inserts the message in to the RIB Hospital tables and message is removed from the JMS topic.

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.

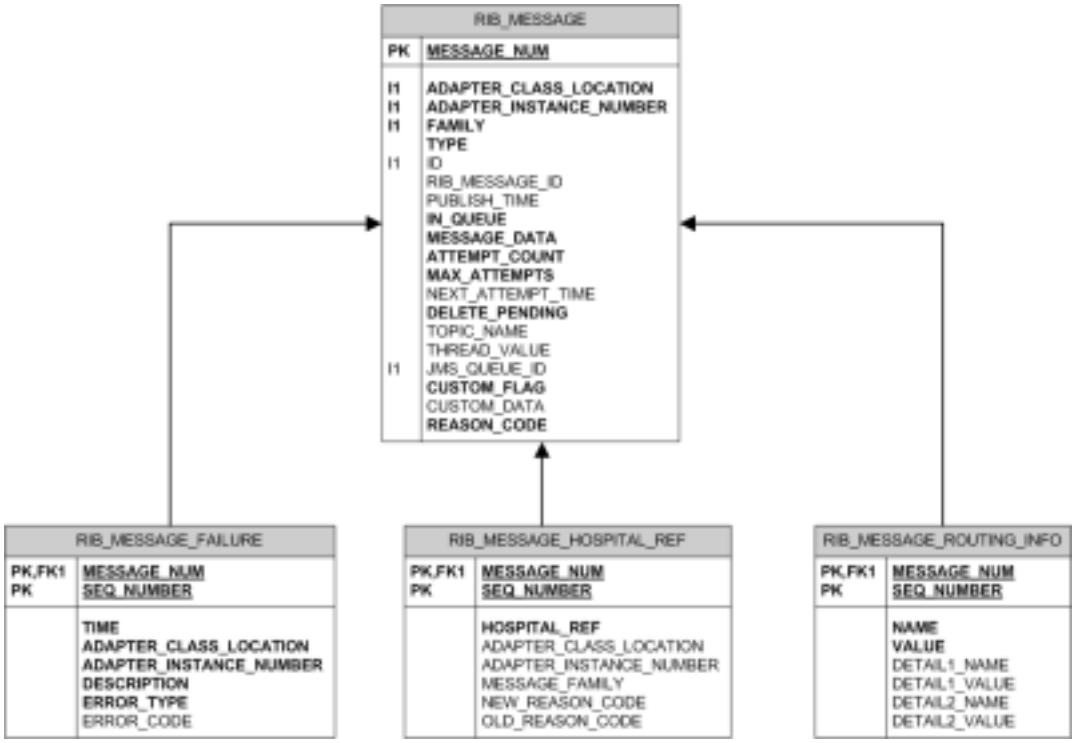
> **Note:**
>
> The XA interface is a standard protocol between a transaction manager and a database or resource manager. Note that both the JMS topic connection and the database connection must support the XA protocol. For more information regarding the XA standard, see the URL `http://www.opengroup.org`.

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 installation (for example, RWMS, SIM, RPM, AIP, RFM, OMS, 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 or what has happened to the record or message information. Each time the message is reprocessed, 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. So manually deleting messages from the hospital database using SQL directly may produce severe processing problems. Also, 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

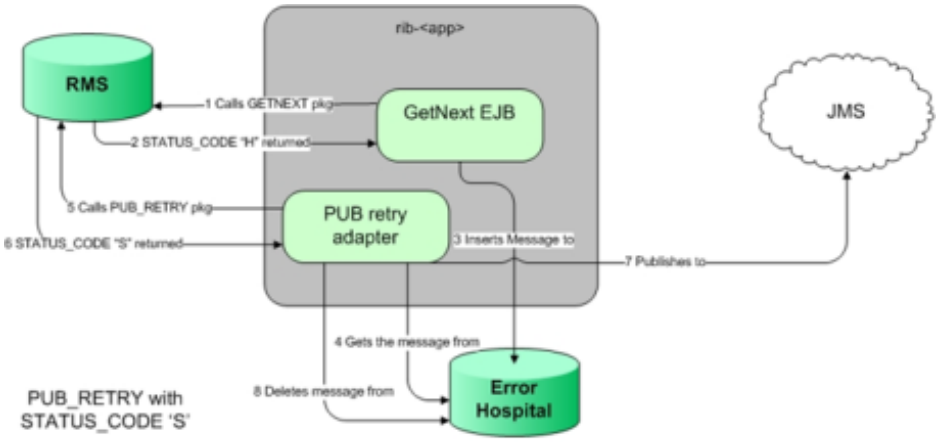
All subscriber side retrying of messages are handled by the Sub Retry Adapter. The Sub Retry Adapter looks at all messages with reason code SUB, then filters and identifies the messages that are ready to be reprocessed, keeping message ordering in mind.

Oracle Retail applications are unaware that the integrations of the business data is happening through a JMS server. RIB abstracts the fact 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.

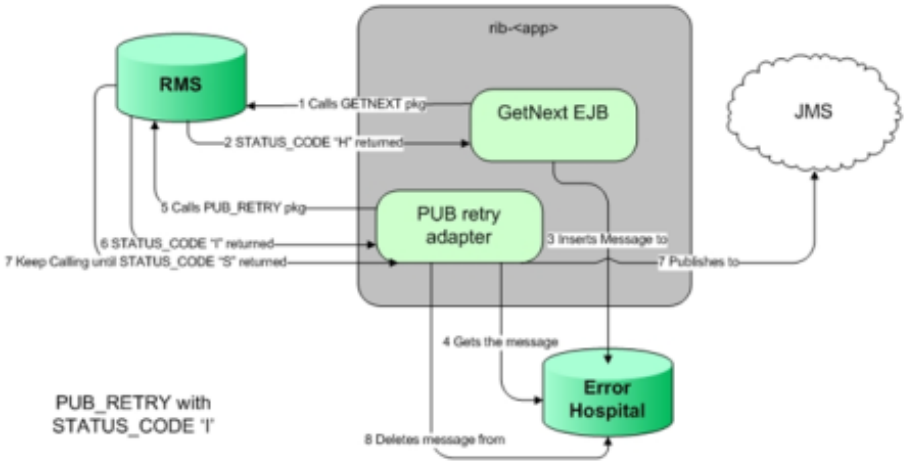
All messages with reason code of PUB are retried by the Pub Retry Adapter. RMS is the only retail application that needs the Pub Retry Adapter.

PUB Retry Adapter

The following diagrams illustrate how the PUB Retry Adapter works.

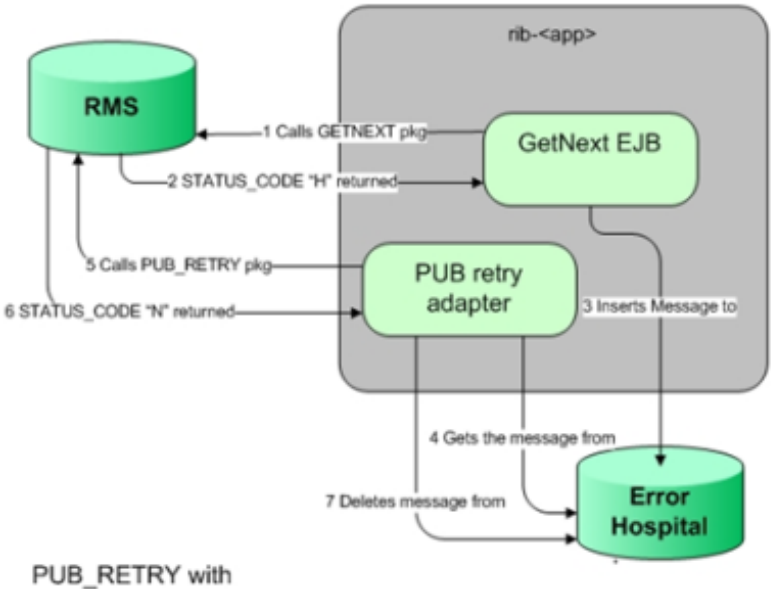


PUB_RETRY with STATUS_CODE 'S'

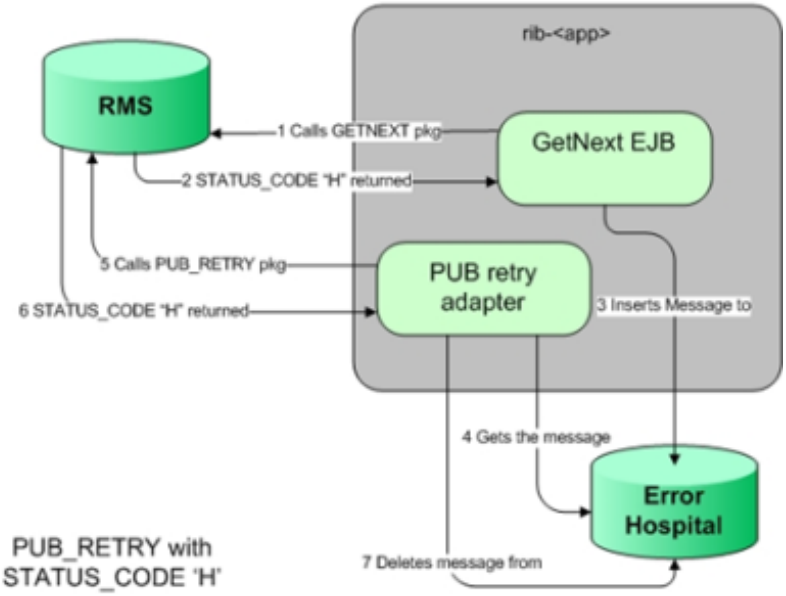


PUB_RETRY with STATUS_CODE 'I'

RIB PUB_RETRY Adapter Processing

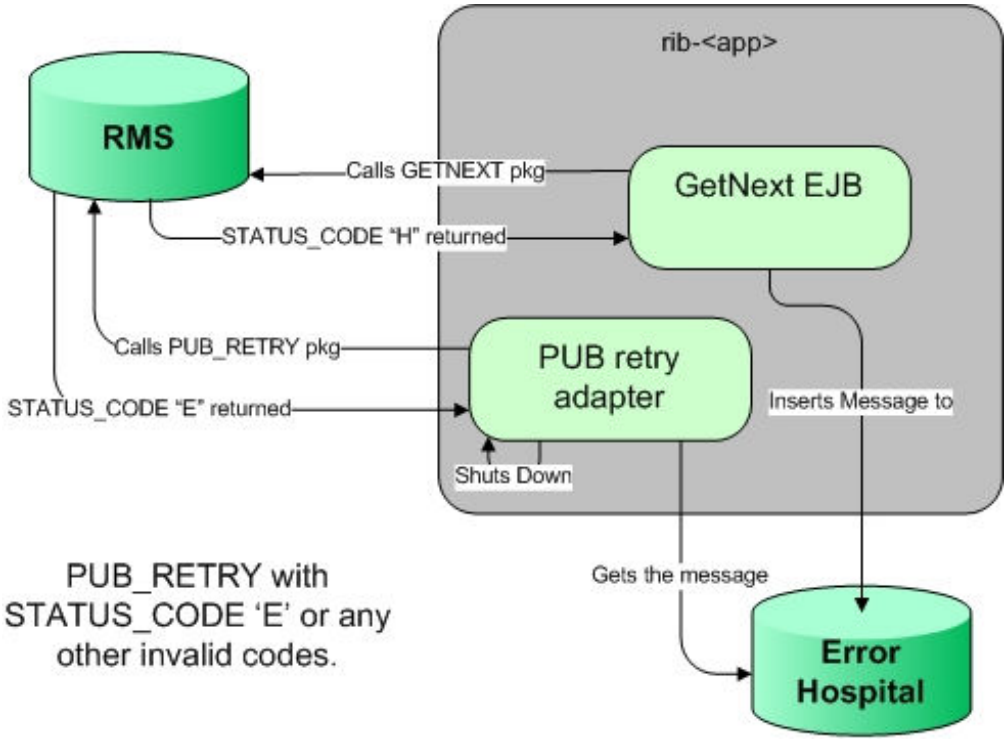


PUB_RETRY with STATUS_CODE 'N'



PUB_RETRY with STATUS_CODE 'H'

RIB PUB_RETRY Adapter Processing



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, a dependency check is 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.

Exception to this rib-tafr app, in case of rib-tafr attempt_count is 1, even if the message is inserted into the hospital as a dependent message because tafr adapters work with two topics and message would already be subscribed once by the tafr, therefore it always has attempt_count=1.

JMS Delivery Count

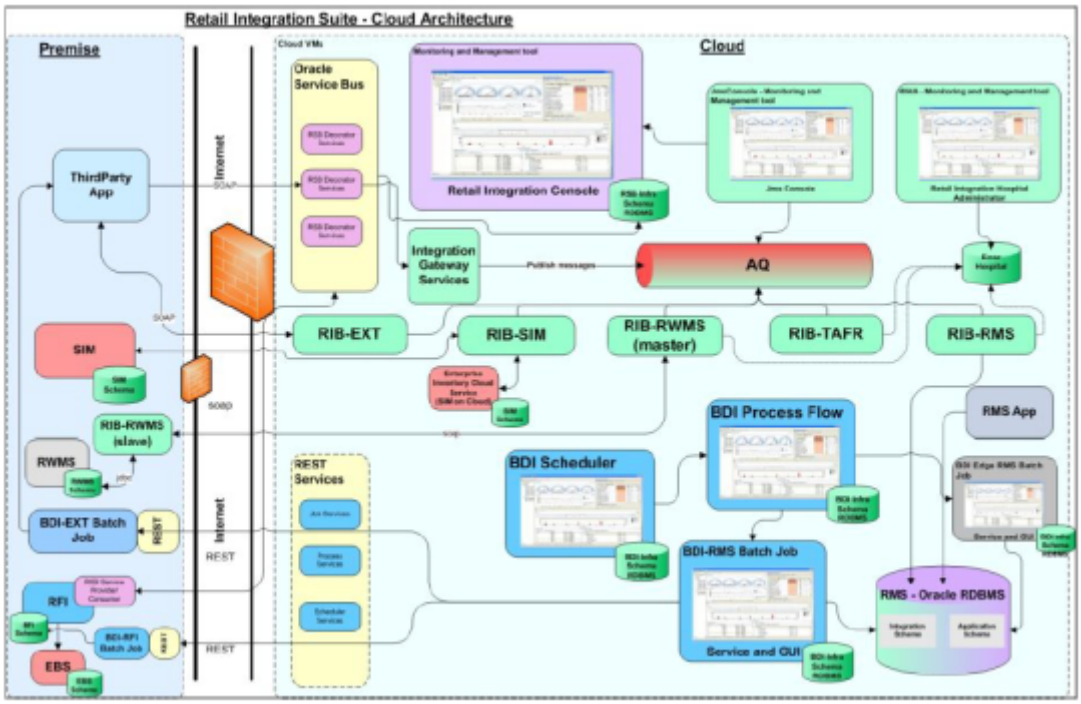
JMSXDeliveryCount is a message property set by AQ JMS. This property is checked to see if the message is being redelivered by the JMS. If the count MAX_REDELIVERY_THRESHOLD (set to 2) is reached, the RIB subscribers assume that the message is being re-delivered; the message will be determined as a poison message. The message is written to the file system (at the same location where application log files are written), and the adapter is shut down in such scenarios. An administrator must decide how this message will be handled.

3
Cloud

This chapter describes the RIB cloud.

The following diagram describes a sample hybrid architecture in which some of the retail applications are on-premise and some other (including RIB) are in the cloud. In this architecture, the retail applications RWMS is on-premise, while RIB is on the cloud.

Figure 3-1 Retail Integration Suite - Cloud Architecture



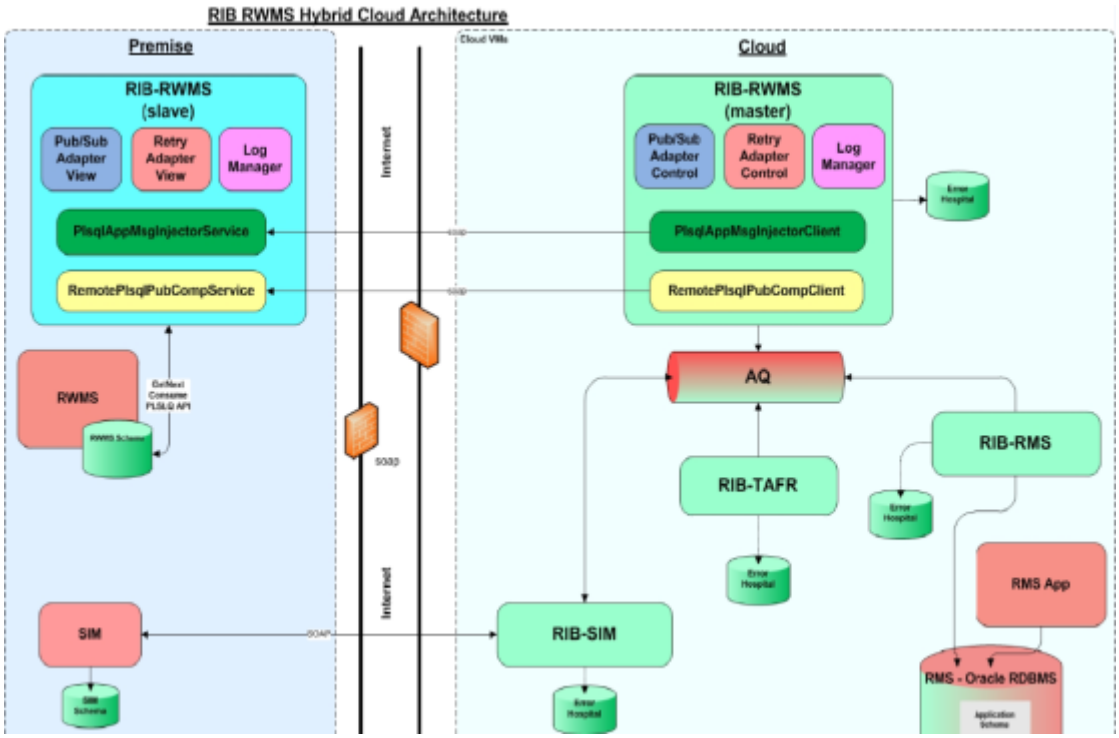
In order to support cloud deployment (including a hybrid cloud), RIB is enhanced with the addition of two Web services. These are injector and publisher Web services that allow retail applications to communicate with other applications.

Configuring RIB-RWMS for Hybrid Cloud Deployment Topology

RWMS on-premise cannot communicate with RMS and other retail apps, which are all in cloud via RIB. As RIB is already supported in cloud, for enabling the integration of RWMS with all other retail applications which are in the hybrid cloud environment, RIB follows the primary/secondary approach. The secondary resides close to on-prem RWMS, while the primary is on-cloud. Communication between primary and secondary is through web service calls. The RIB-RWMS primary invokes the new web services exposed by secondary RIB-RWMS to send/receive messages to/from other applications on cloud via RIB.

For RIB-RWMS to communicate with RWMS on premise and RIB on cloud, it should be deployed in primary-secondary topology. Hybrid cloud set-up for RWMS involves a two part installation, one for each primary (cloud) and secondary components (on-premise).

Figure 3-2 RIB-RWMS Hybrid Cloud Architecture



> **Note:**
>
> The client-server architecture is only applicable to RIB and RWMS integration, where RIB is deployed on Next Gen SaaS Platform and the legacy RWMS is hosted on on-prem/PaaS.

Installation and Setup instructions for RIB-RWMS Secondary (On-Premise)

This section describes the installation and setup instructions. This includes the installation prerequisites, preparing the WebLogic server, creating a WebLogic domain, verifying installation of wls policies, extending an existing domain to add wls policies, and deploying the EAR file.

> **Note:**
>
> The screen captures included in the following steps are for example only. Therefore, consider the illustrations as guides only; the values shown may not always apply.

Installation Prerequisite

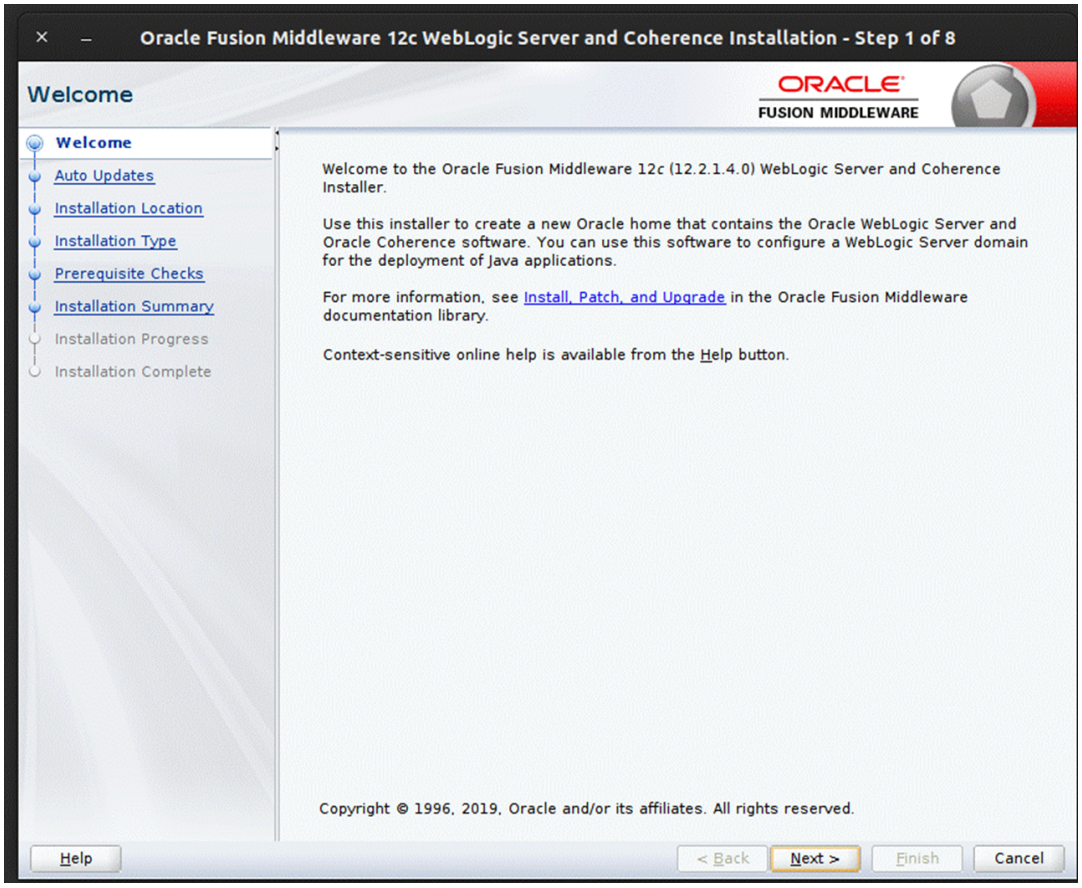
The `rib-rwms` secondary (on-premise) application requires Oracle WebLogic Server 12c (12.2.1.4.0) and must be built with Java 8 (JDK 1.8.0+ 64 bit or later), with the latest security updates.

> **Important:**
>
> If there is an existing WebLogic 12.x.x or 10.3.xc installation on the server, you must upgrade to WebLogic 12.2.1.4.0. All middleware components associated with WebLogic server 10.3.6 should be upgraded to 12.2.1.4.0. Back up the `weblogic.policy` file (`$WLS_HOME/wlserver/server/lib`) before upgrading your WebLogic server, because this file could be overwritten. Copy over the `weblogic.policy` backup file after the WebLogic upgrade is finished and the post-patching installation steps are completed. For upgrading your WebLogic server to 12.2.1.4.0, use the appropriate Upgrade Installer.

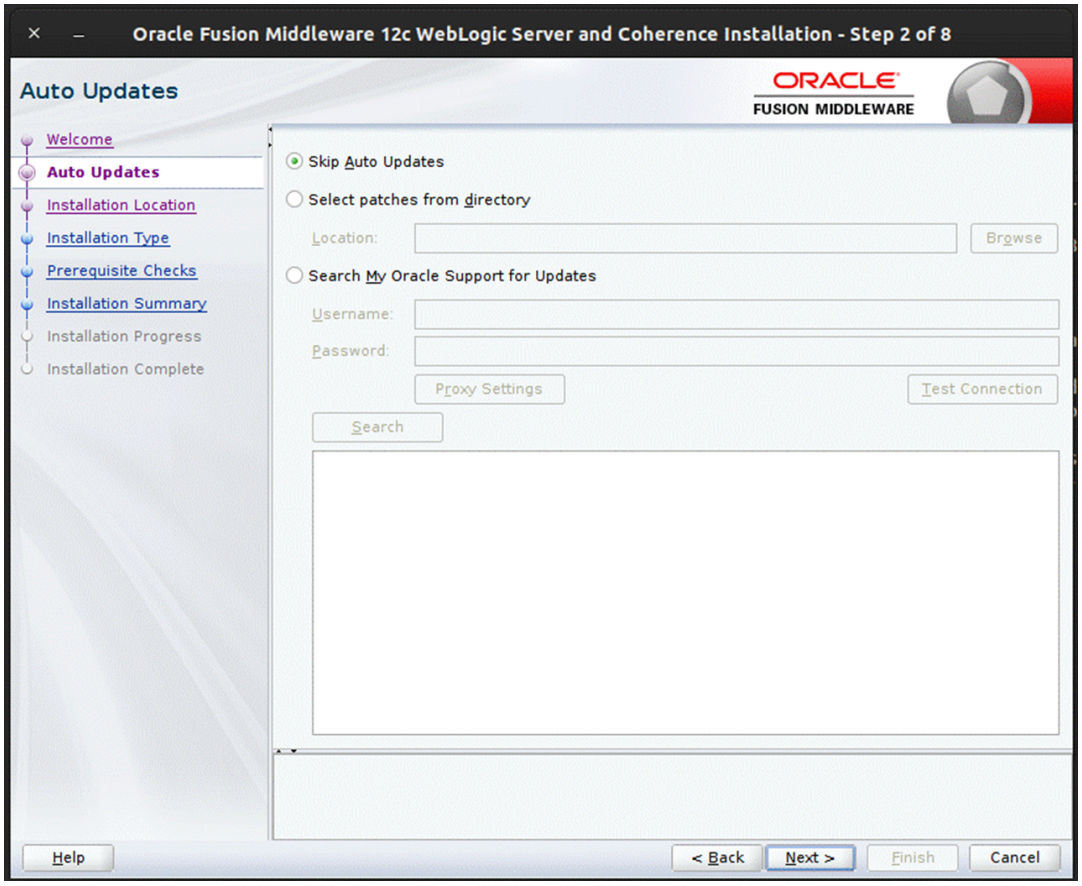
Prepare the WebLogic Server

Take the following steps to prepare the WebLogic server:

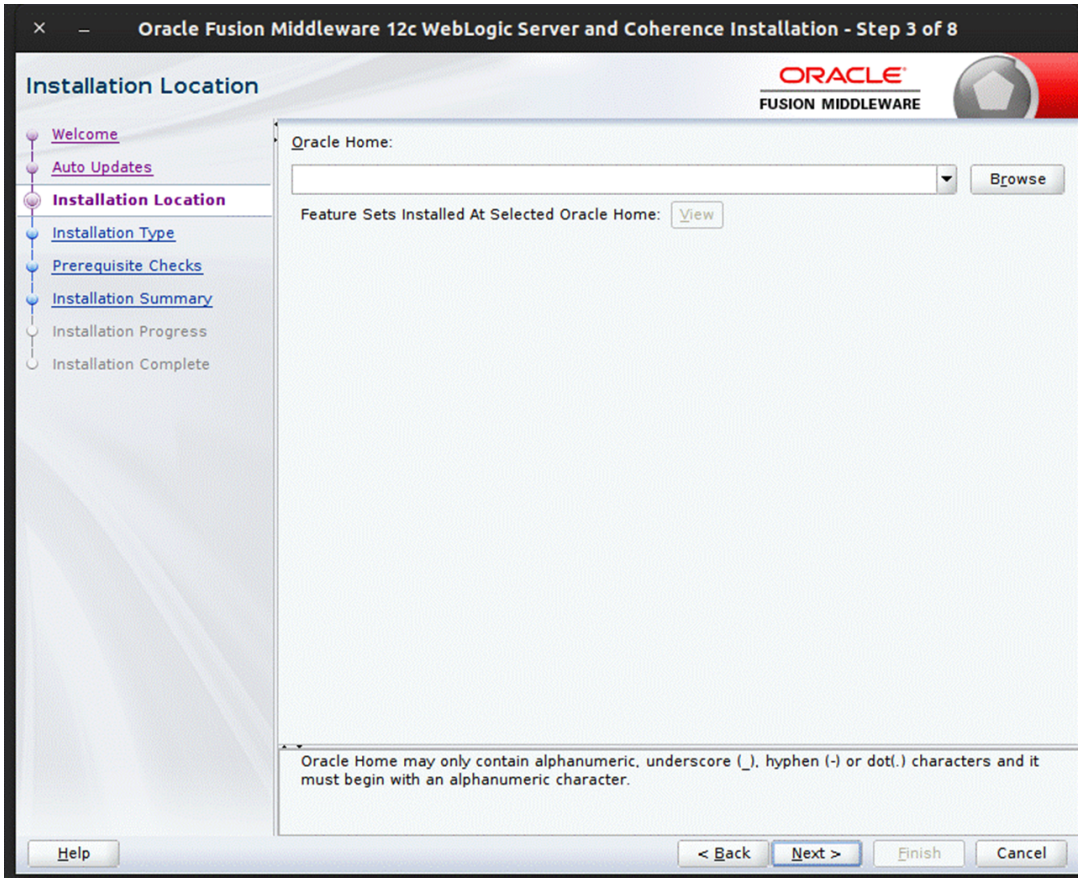
1. Find `fmw_12.2.1.4.0_infrastructure_Disk1_1of1.zip` and download this file to your system.
2. Extract the contents of this zip file to your system. Use the `fmw_12.2.1.4.0_infrastructure.jar` file to run the installer.
3. Run the installer by executing the `java -jar fmw_12.2.1.4.0_infrastructure.jar` file. The Welcome window displays.



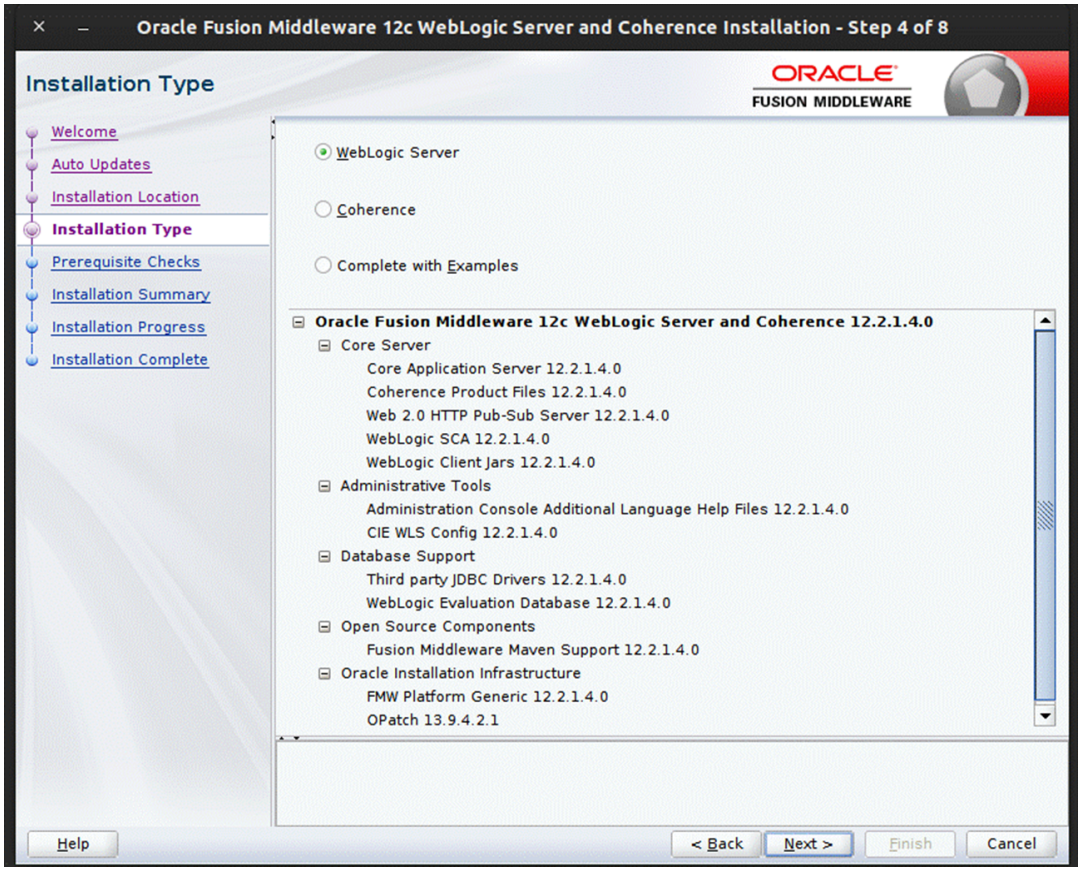
4. Click **Next**. The Auto Updates window displays.



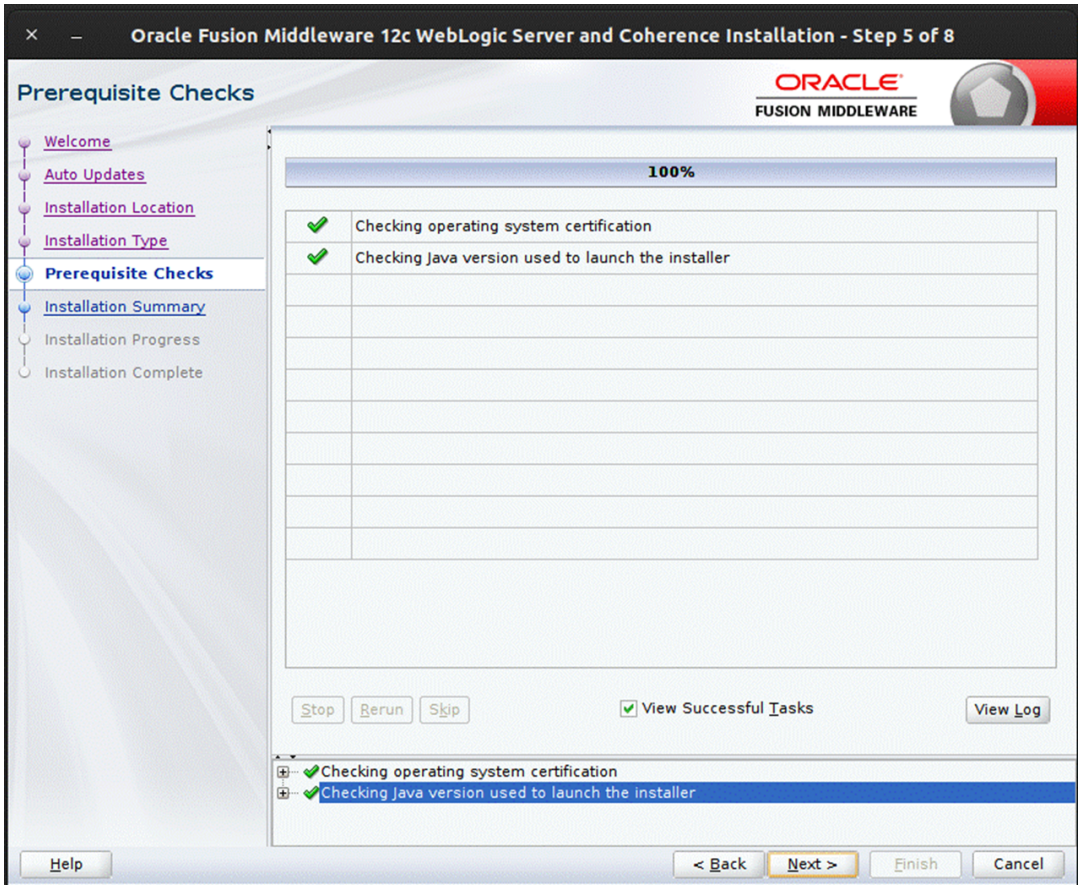
5. Select the appropriate radio button and click **Next**. The Installation Location window displays.



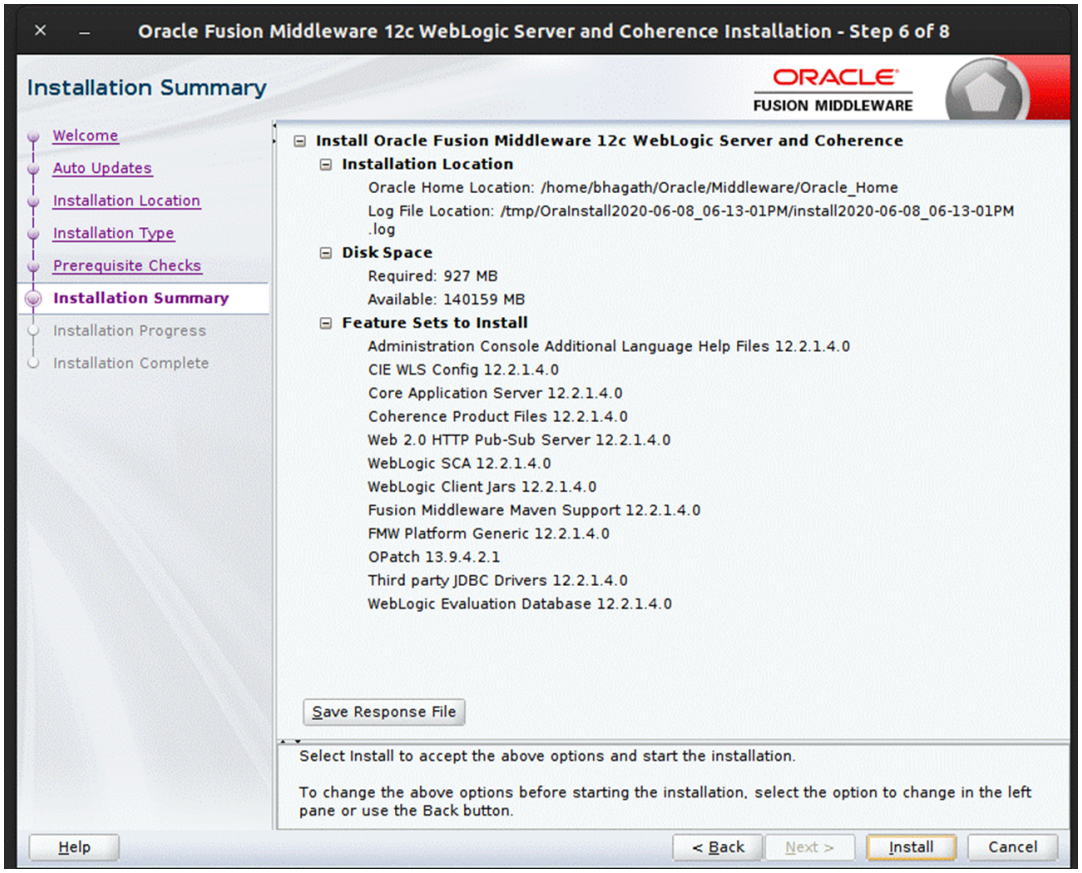
6. Click **Browse** to select the Oracle Home location where the Weblogic server is to be installed. Click **Next**. The Installation Type window displays.



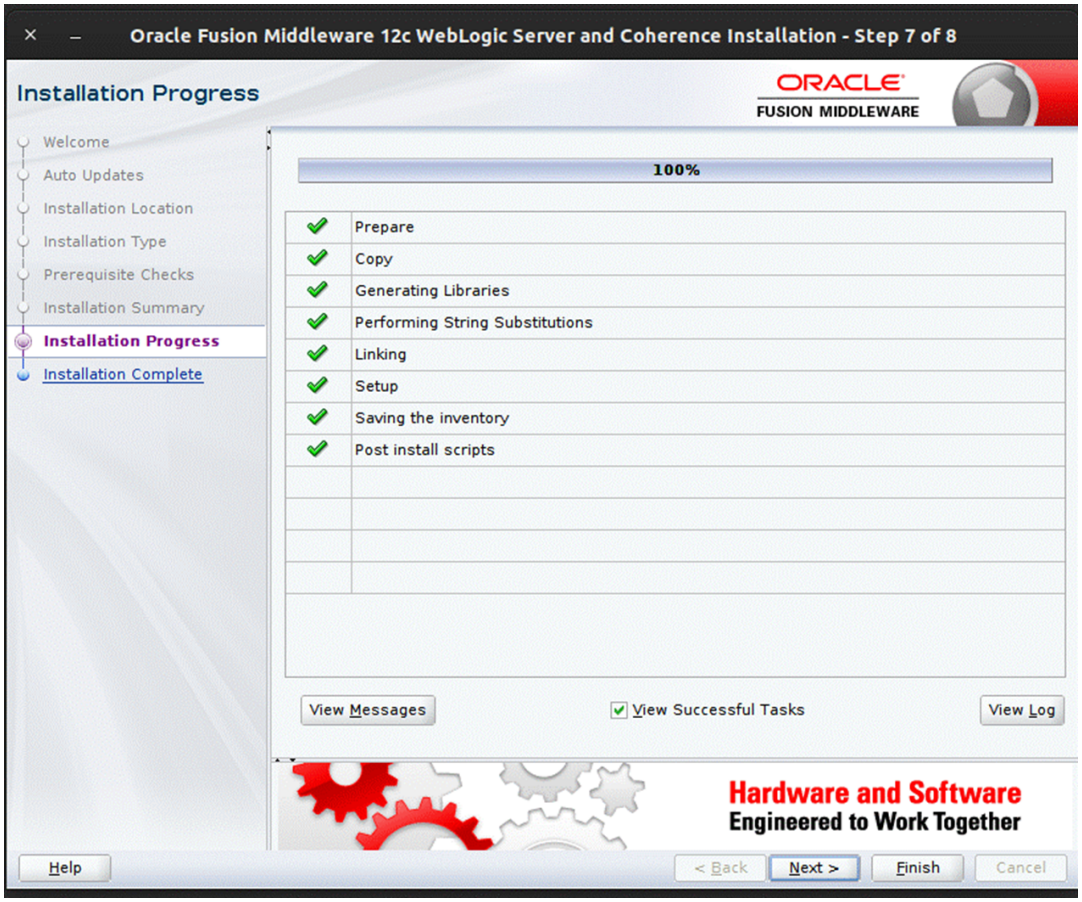
7. Select **Fusion Middleware Infrastructure** and click **Next**. The installer performs the prerequisite checks and ensures all required conditions are satisfied.



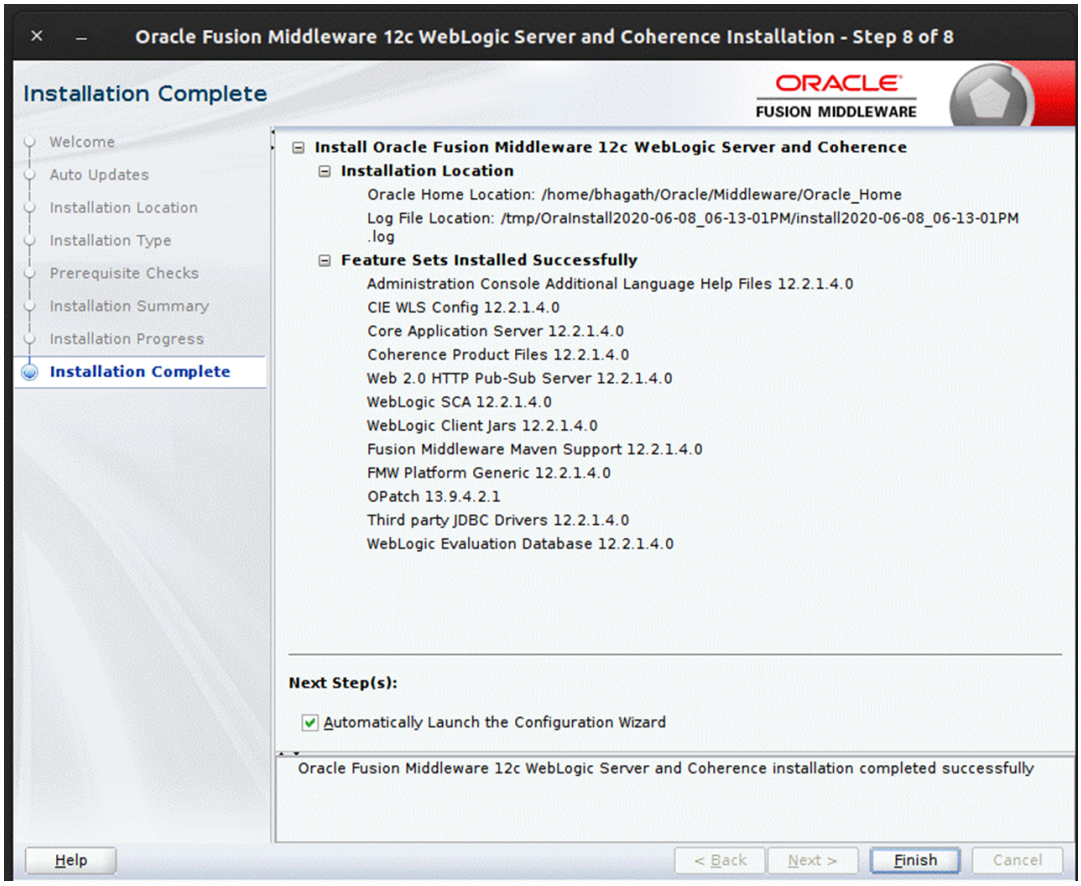
8. When the prerequisite check completes successfully, click **Next**. The Installation Summary window displays.



9. Click **Install**. The Installation Progress window displays.



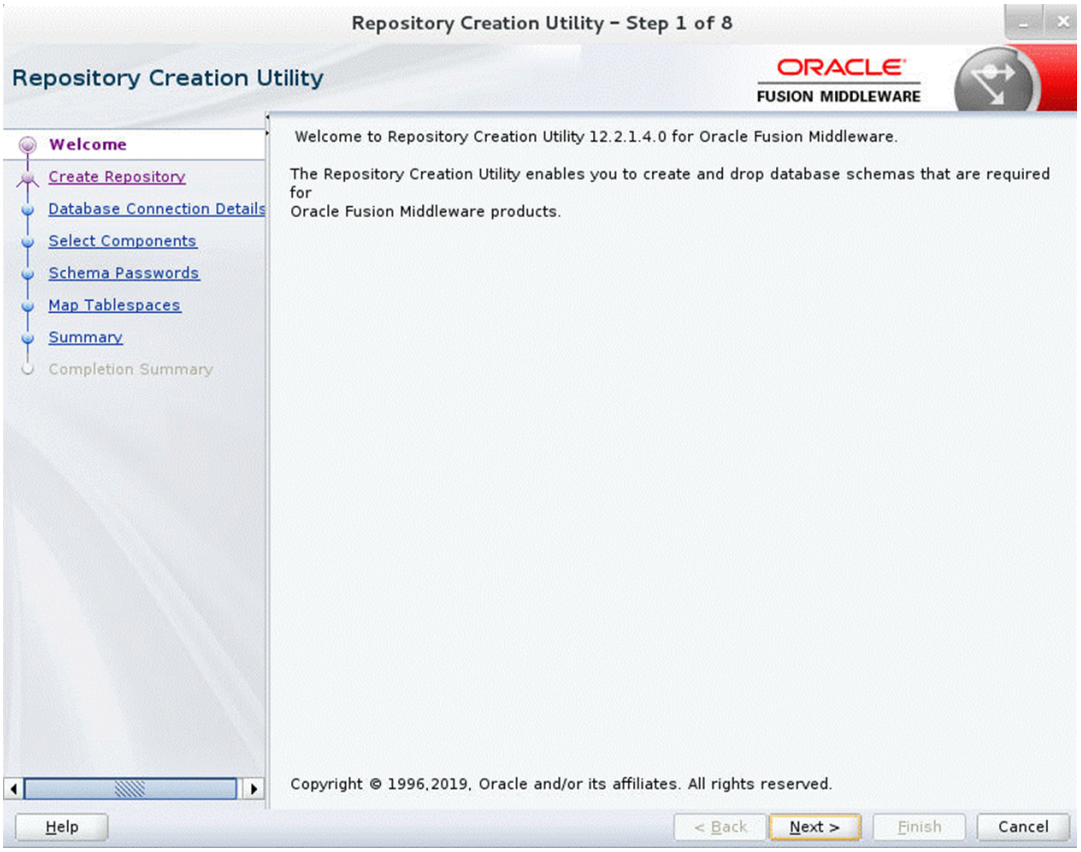
10. Click **Next** when the installation completes. The Installation Complete window displays.



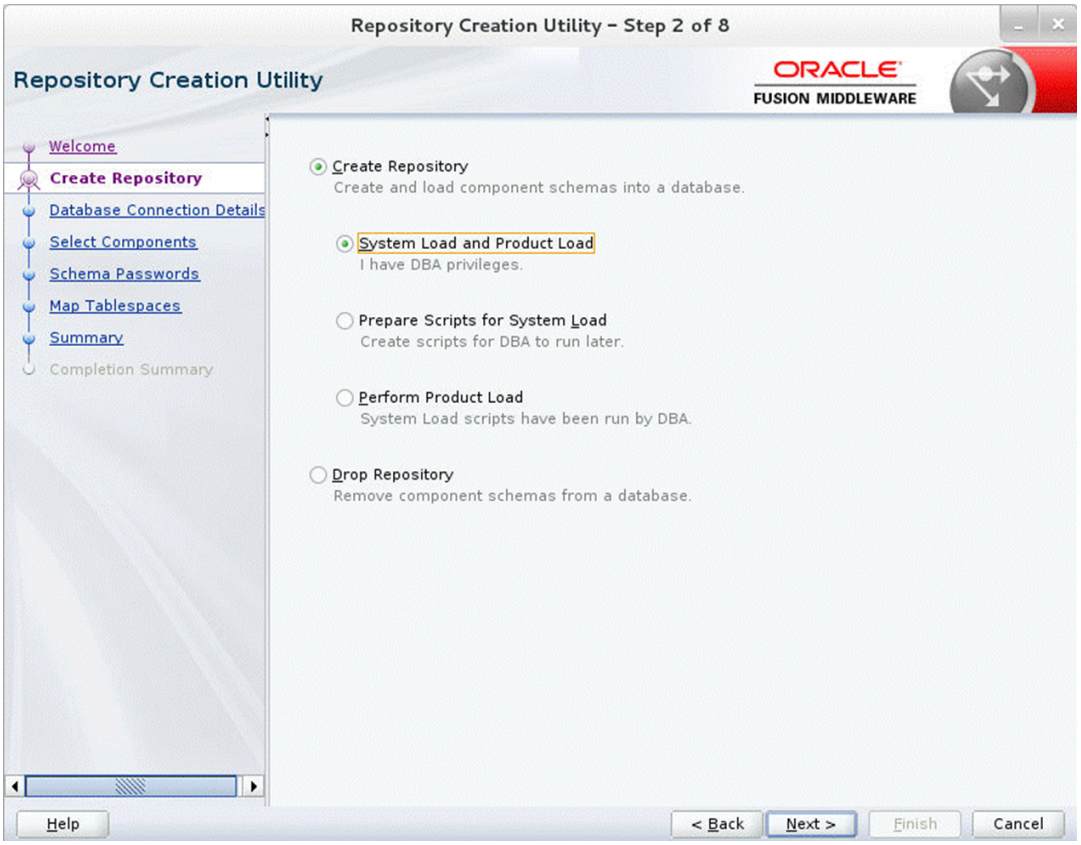
Creating Required RCU Schema Using the Repository Creation Utility

To create a schema user for the domain, take the following steps:

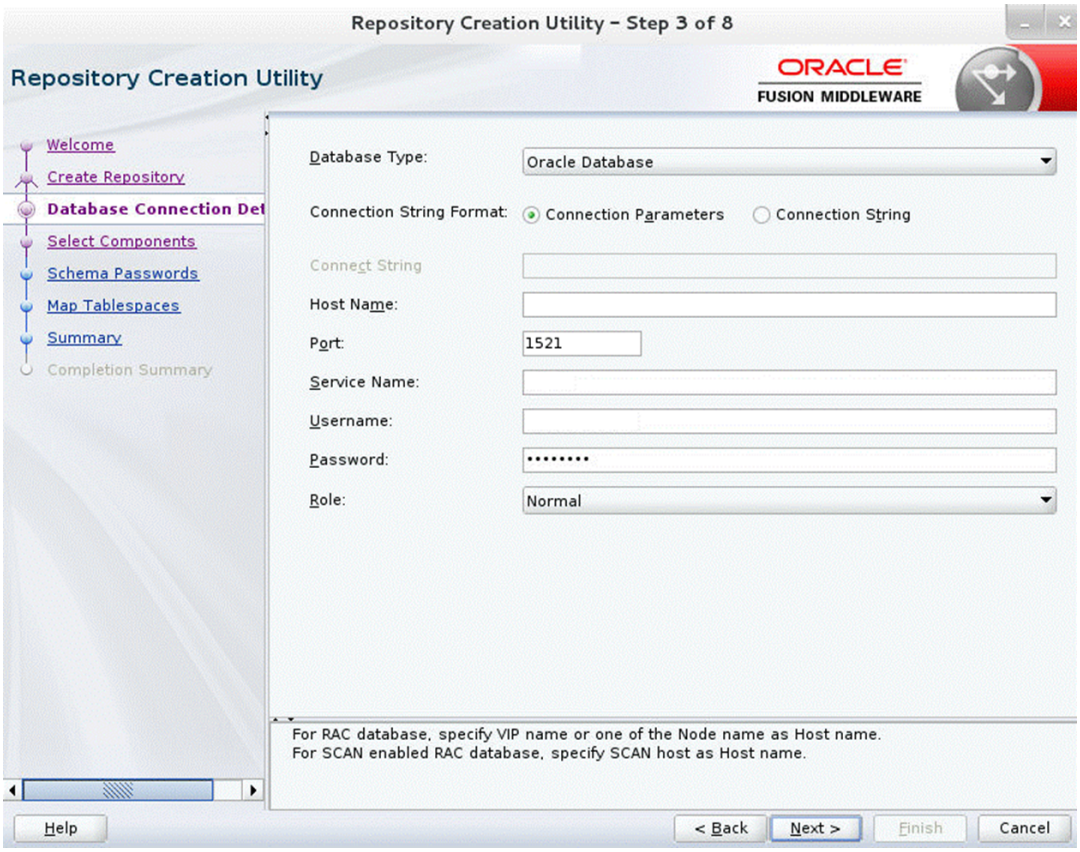
1. Run the RCU from the `<MW_HOME>/oracle_common/bin` folder. The Welcome window displays.



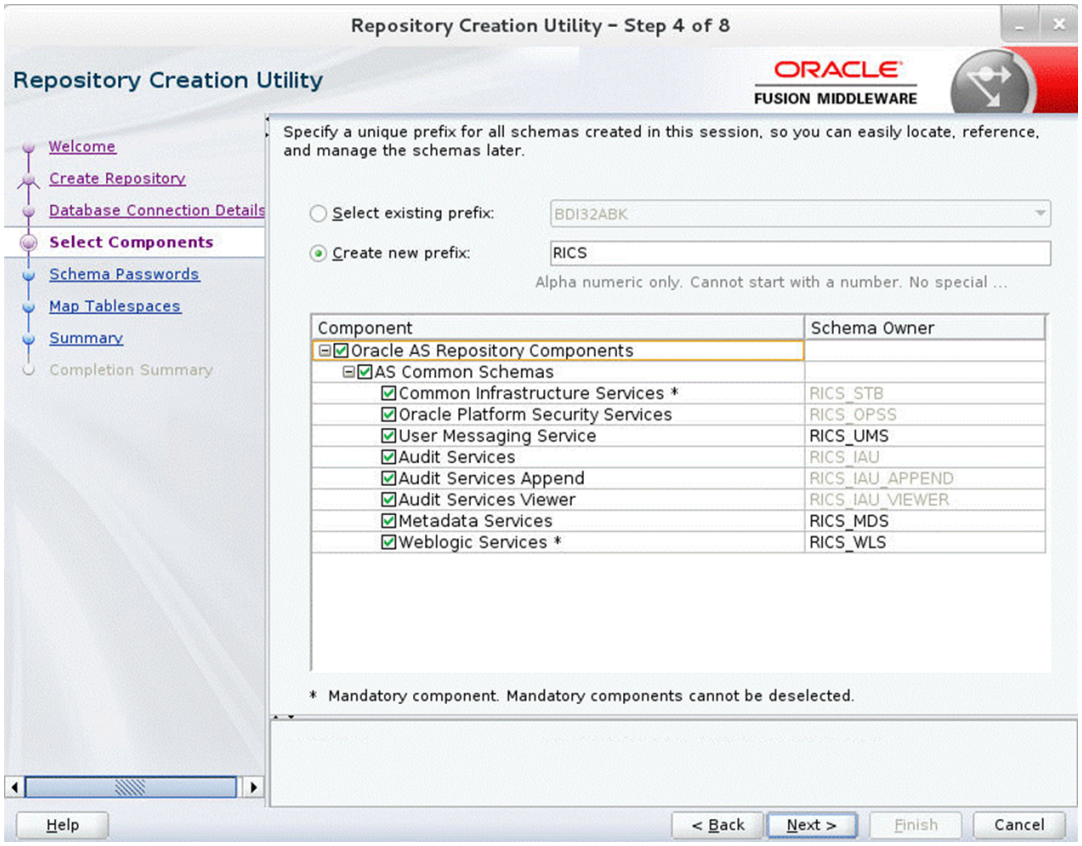
2. Click **Next** and select the **Create Repository** option.



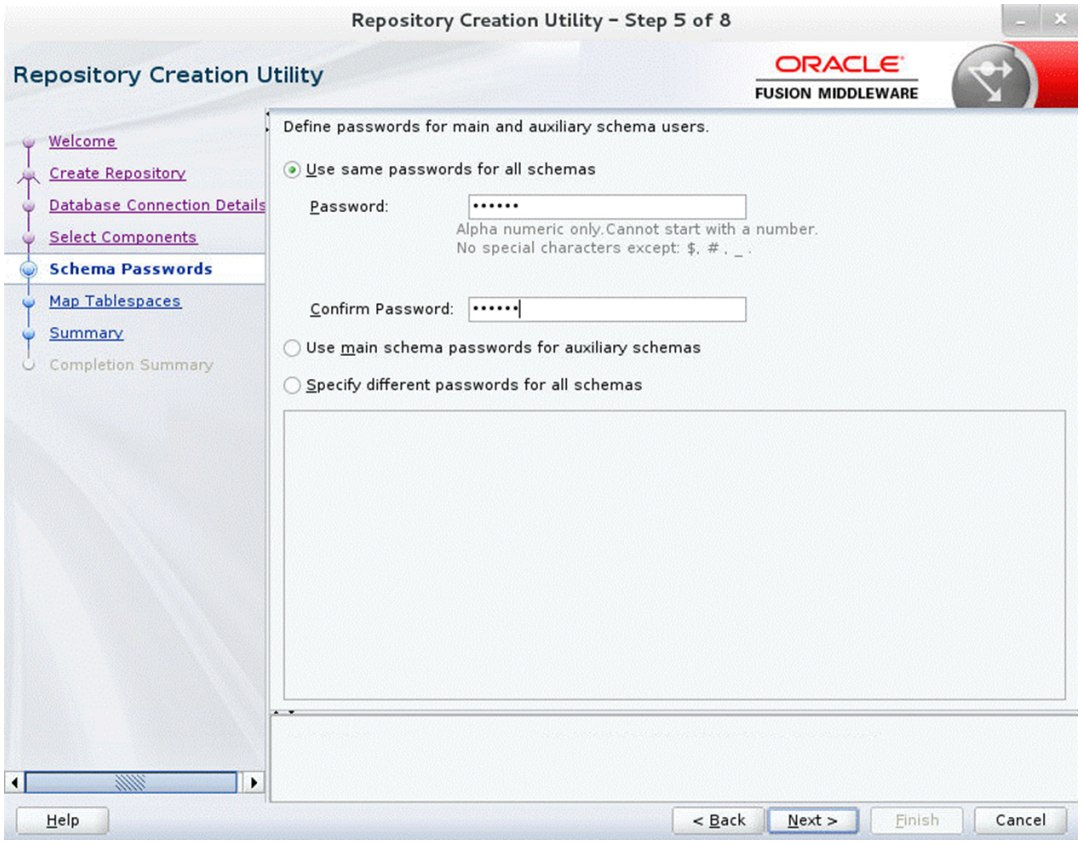
3. Click **Next**. Enter the database credentials where the schema user has to be created.



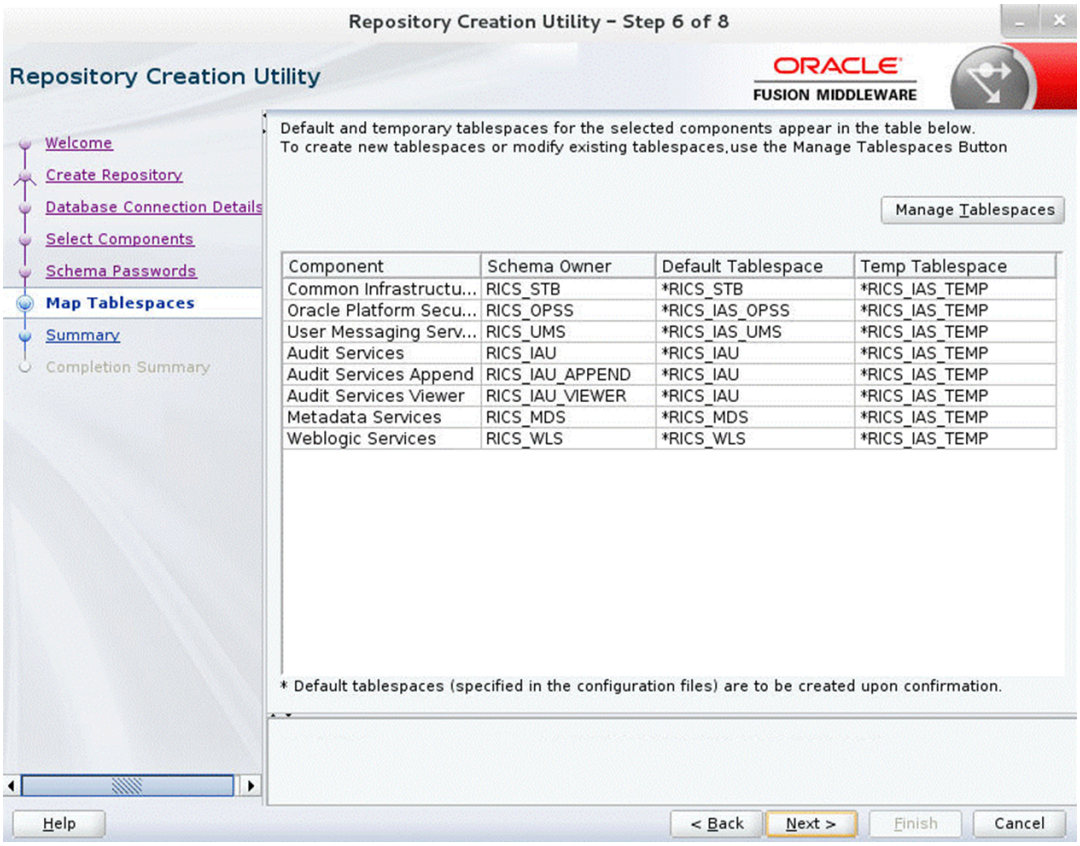
4. Click **Next**. Specify the prefix to be used for the schema user creation. For example, `INT`. Select **Metadata Services**, **Weblogic Services**, and **Oracle Platform Security Services**.



5. Click **Next**. Specify the password.



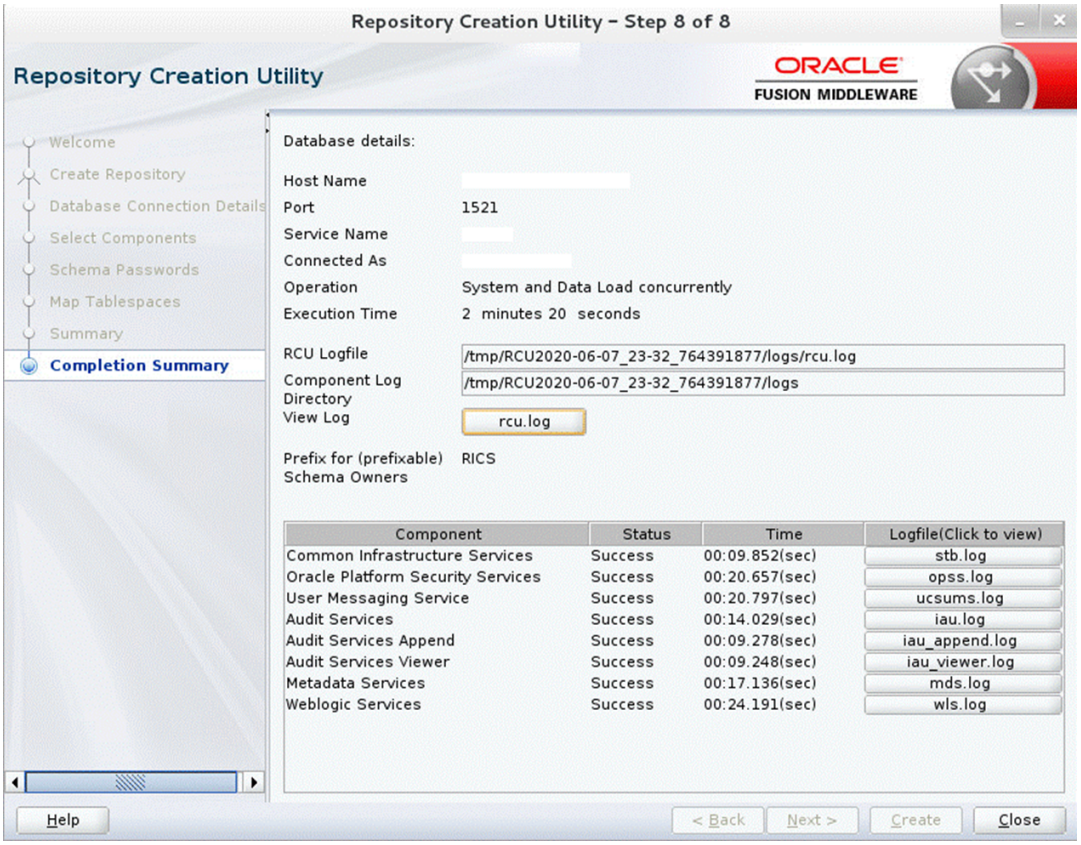
6. Click **Next**. The window provides the details of tablespaces created as part of schema creation.



7. Click **Next**. The Confirmation window displays.

8. Click **OK**. The Summary window displays.

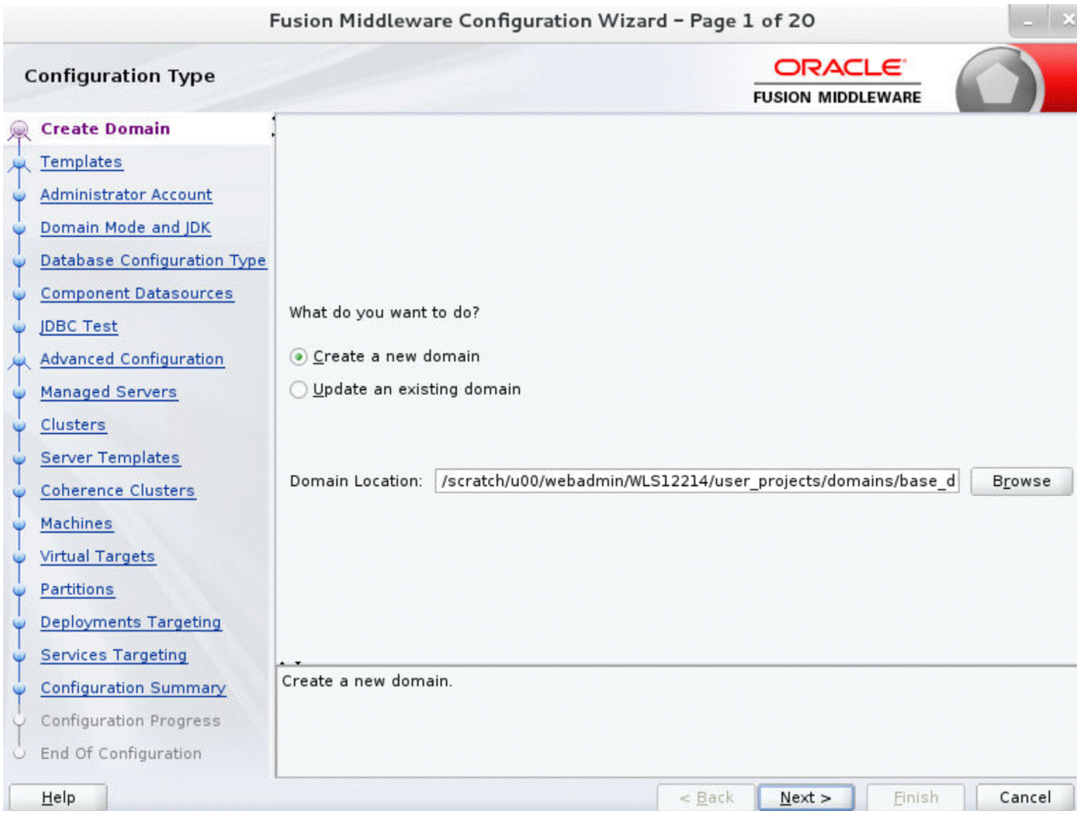
9. Click **Create and proceed** to create the schema. This could take a while to complete. The Completion Summary window displays.



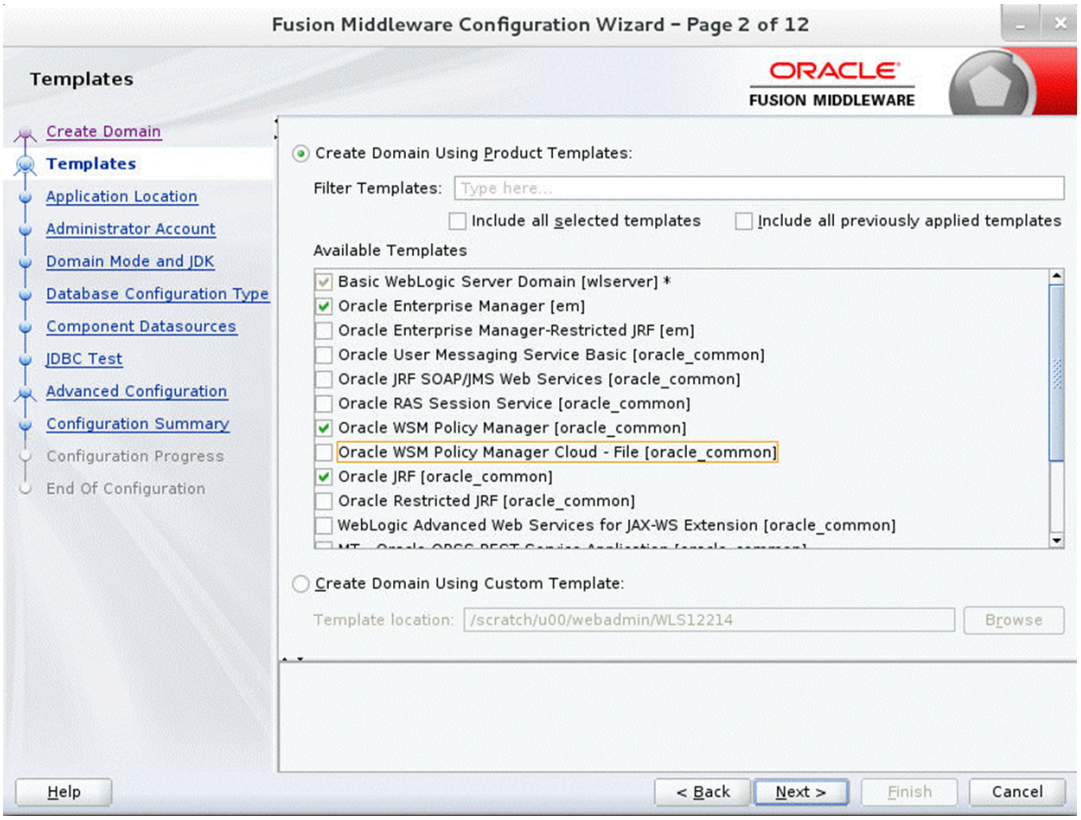
Creating a WebLogic Domain with wls Policy

To create a new WebLogic domain with wls policy, take the following steps:

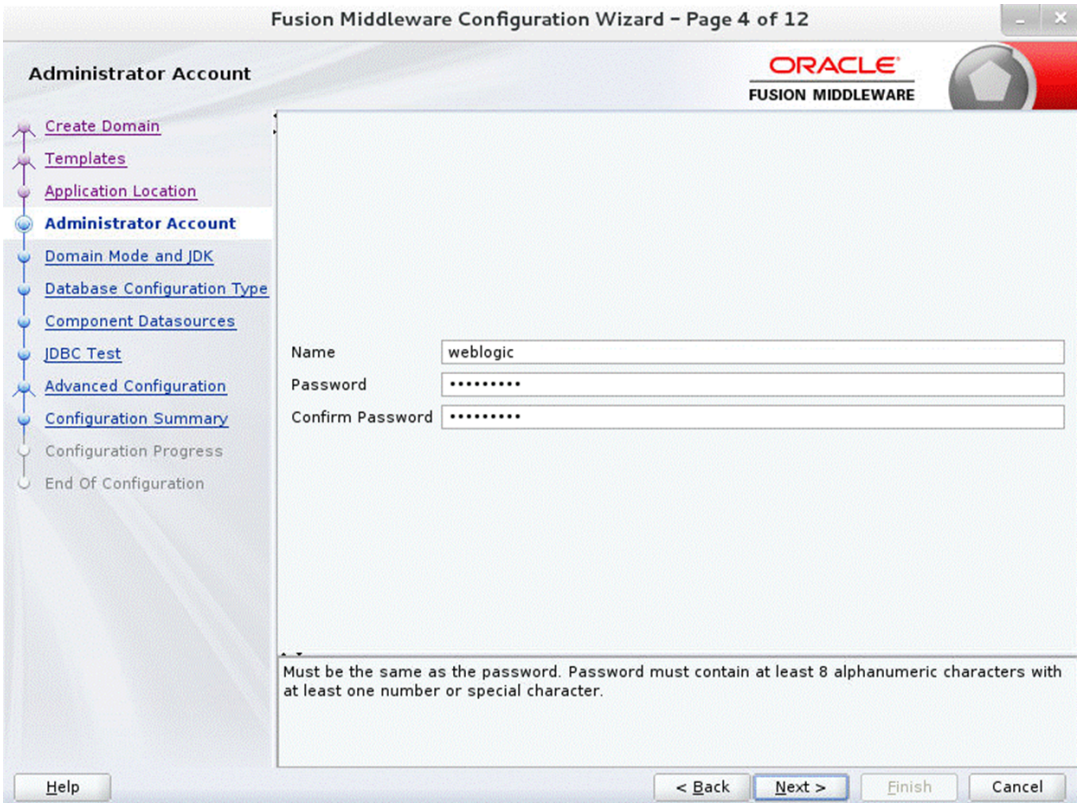
1. Run `config.sh` from the `<ORACLE_HOME>/oracle_common/common/bin` folder. The Configuration Type window displays.



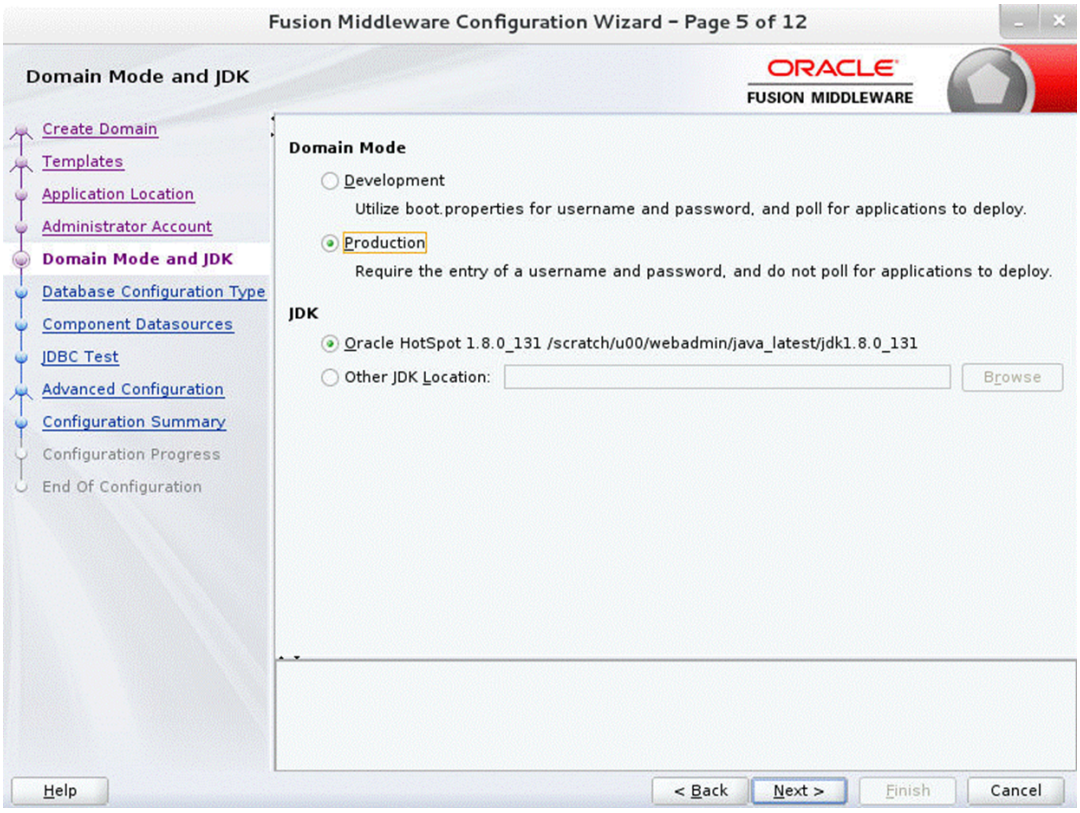
2. Select **Create a new domain**, provide **Domain Location**, and click **Next**. The Templates window displays. By default, the **Basic WebLogic Server Domain [wlserver]** checkbox is selected. Select the **Oracle JRF [oracle_common]**, **Oracle Enterprise Manager [em]**, **Oracle WSM Policy Manager [oracle_common]**, and **Weblogic Advanced WebServices for JAX-WS Extension [oracle_common]** check boxes.



3. Click **Next**. The Application location window displays; provide the application location.
4. Click Next. The Administrator Account window displays. Enter the user credentials you want to use to log in to the WebLogic Administration Console.

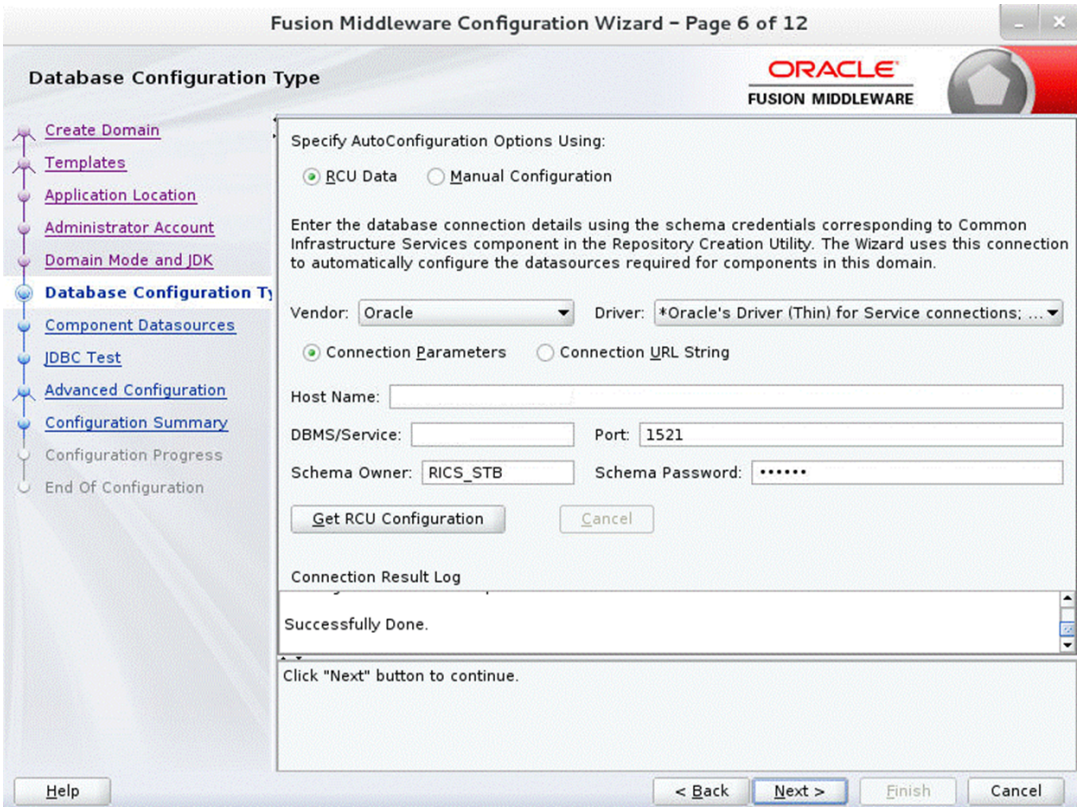


5. Click **Next**. The Domain Mode and JDK window displays. Set the **Domain Mode** as **Production** and select the JDK version (JDK 1.8 with the latest security updates) you want to use.

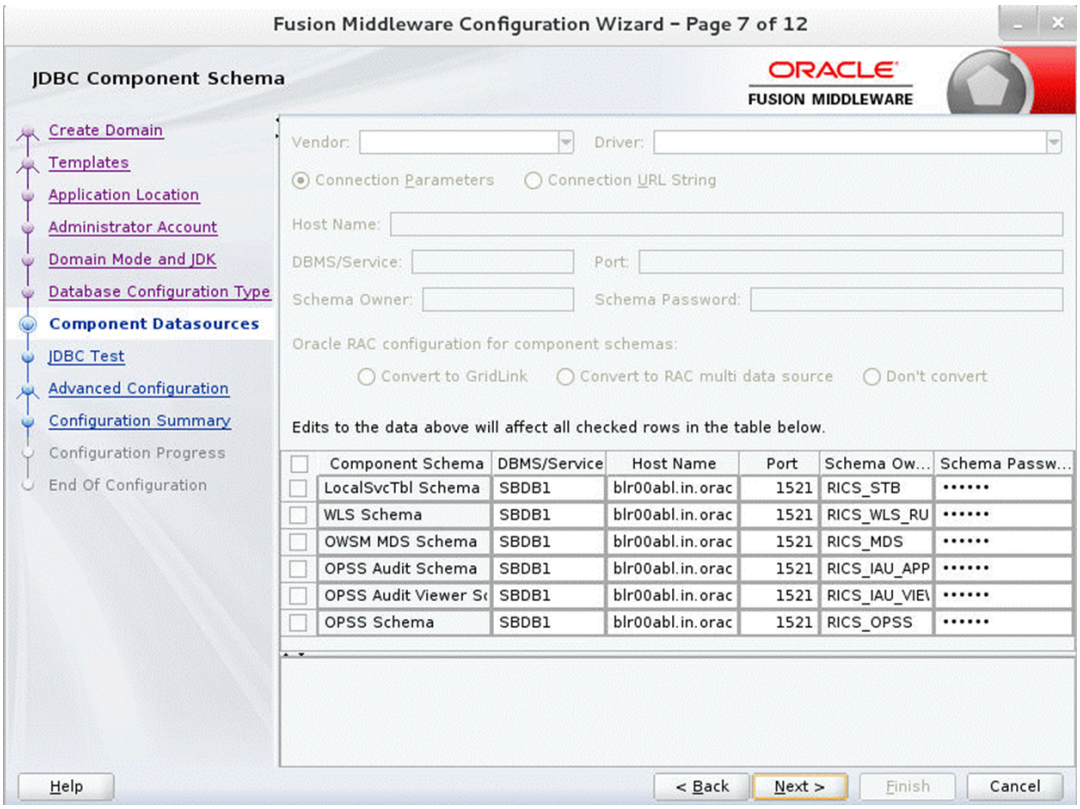


6. Click **Next**. The Database Configuration Type window displays.
   a. Select the **RCU Data** radio button.
   b. Select **Oracle** as the **Vendor**.
   c. Select **Oracle's Driver (Thin) for Service connections; Version 9.0.1 and later** as the **Driver**.
   d. Enter the **Service**, **Host Name**, **Port**, **Schema Owner**, and Schema Password for the `*_STB` schema created using RCU.
   e. Click **Get RCU Configuration**.

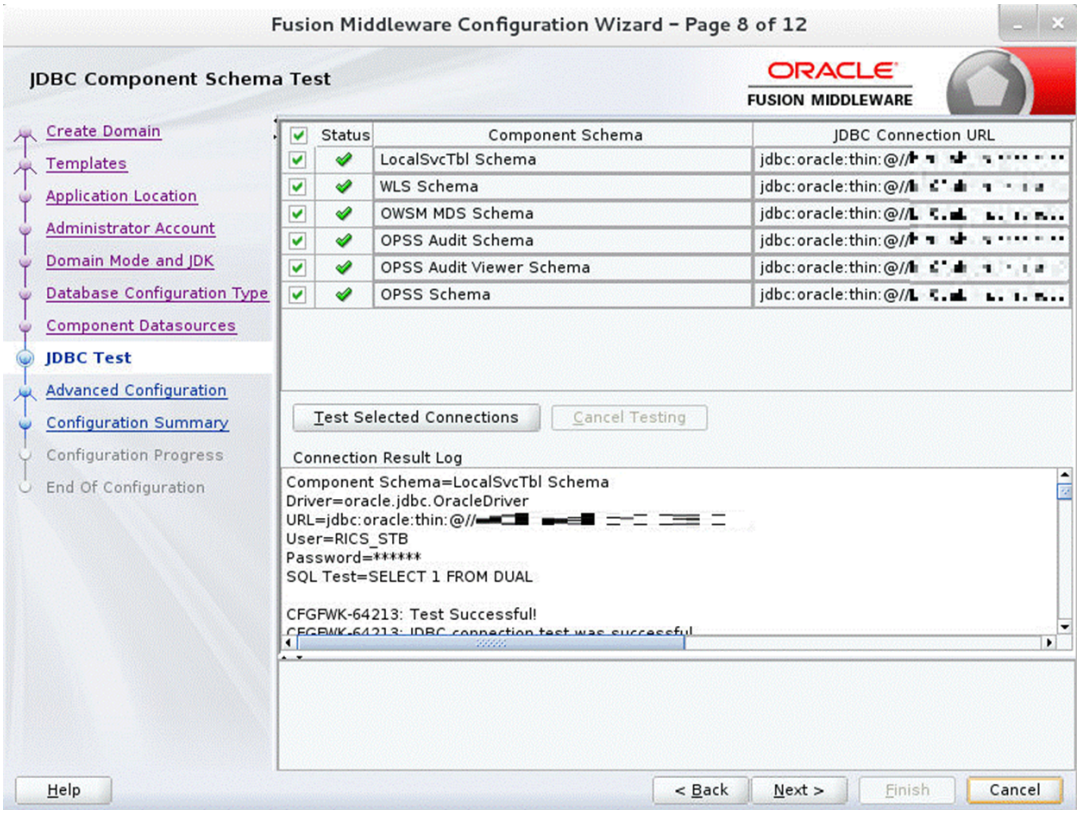
   The Connection Result Log displays the connection status.



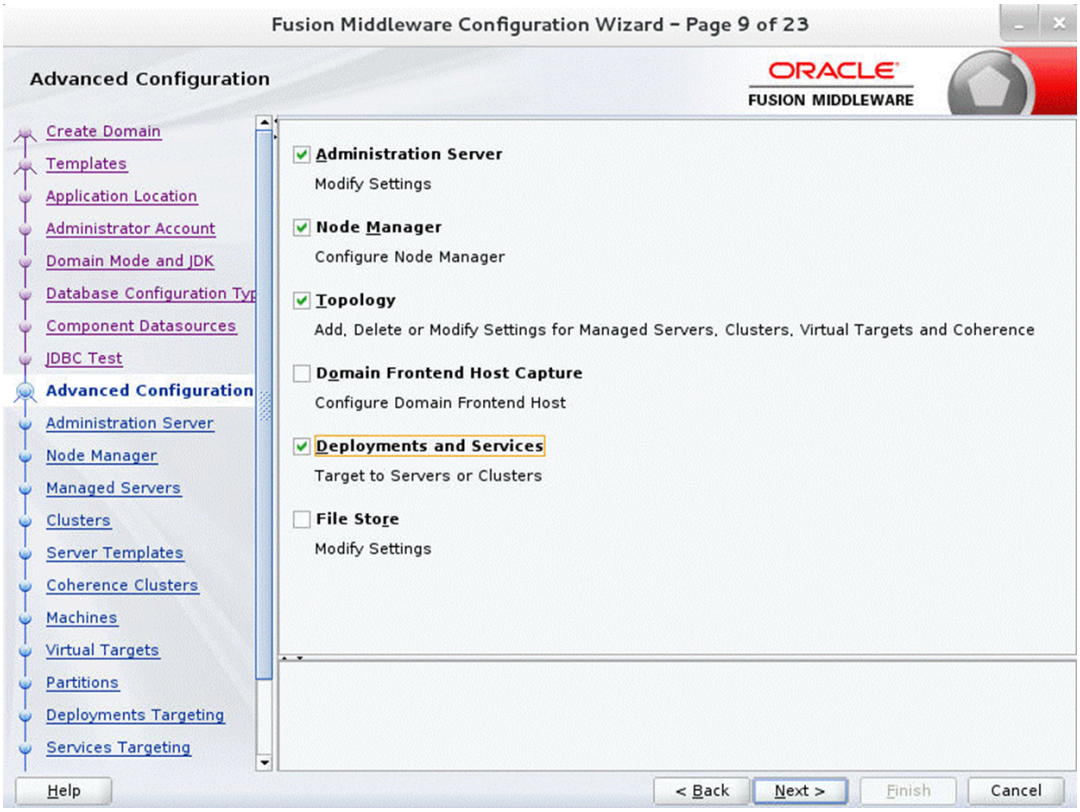
7. Click **Next**. The JDBC Component Schema window displays.



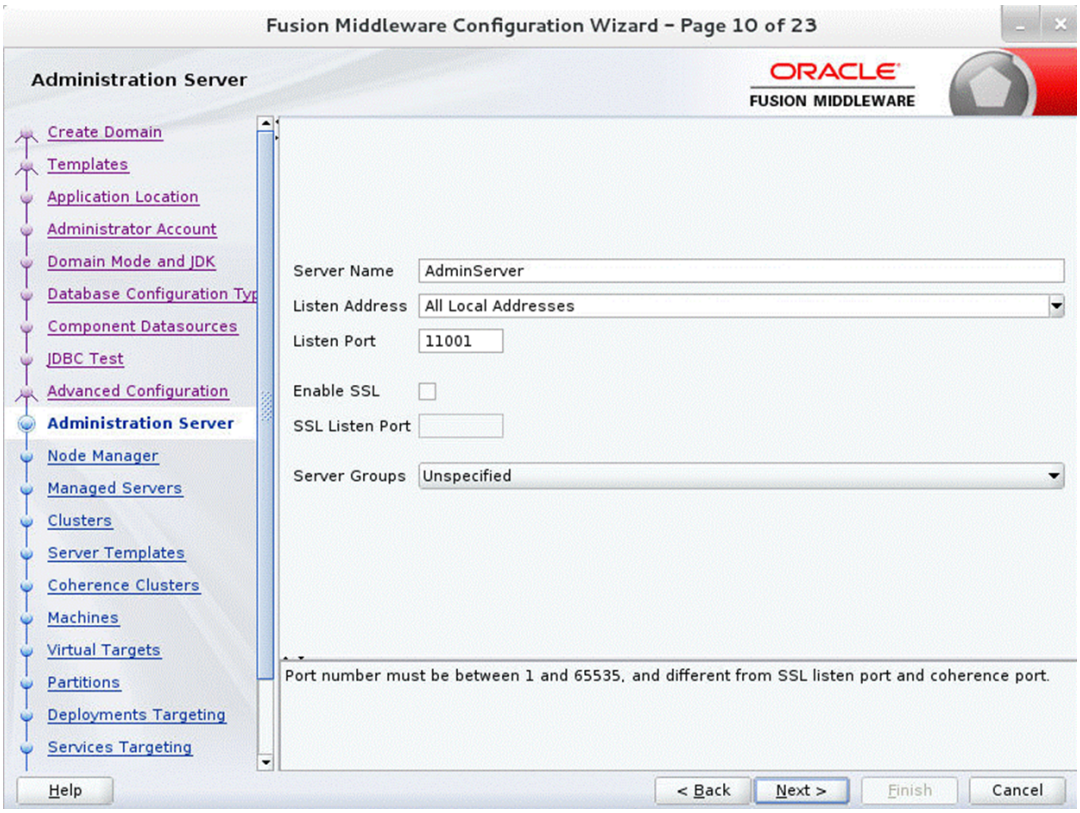
8. Click **Next**. The JDBC Component Schema Test window displays the status on whether the JDBC tests on the schemas were successful.



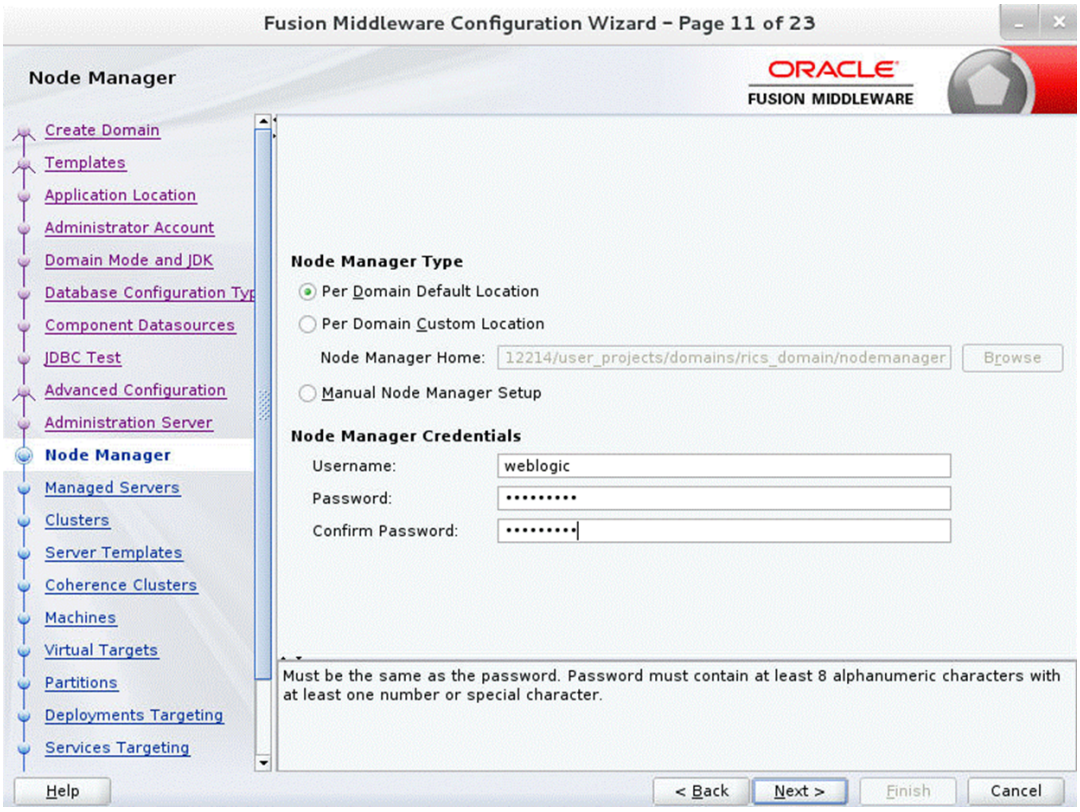
9. Click **Next**. The Advanced Configuration window displays. Select all the checkboxes, except the **Domain Frontend Host Capture** and JMS **File Store** options, in this window.



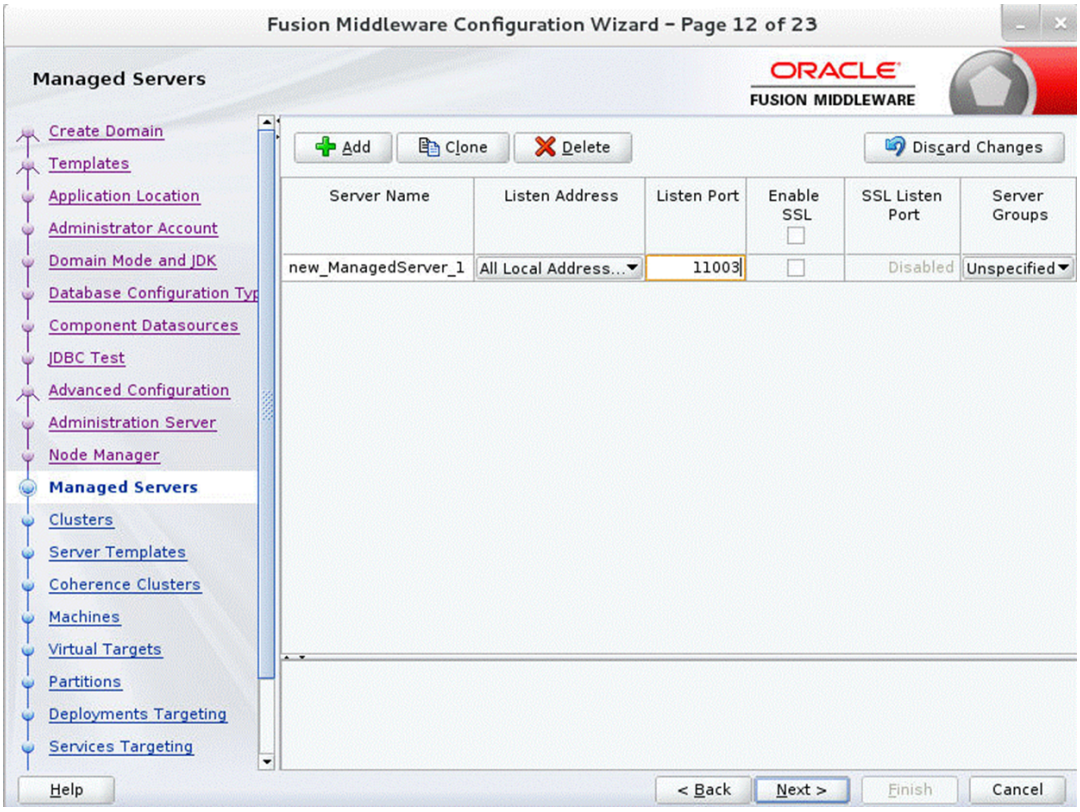
10. Click **Next**. The Administration Server window displays. Enter the Listen Address and the Listen Port details.



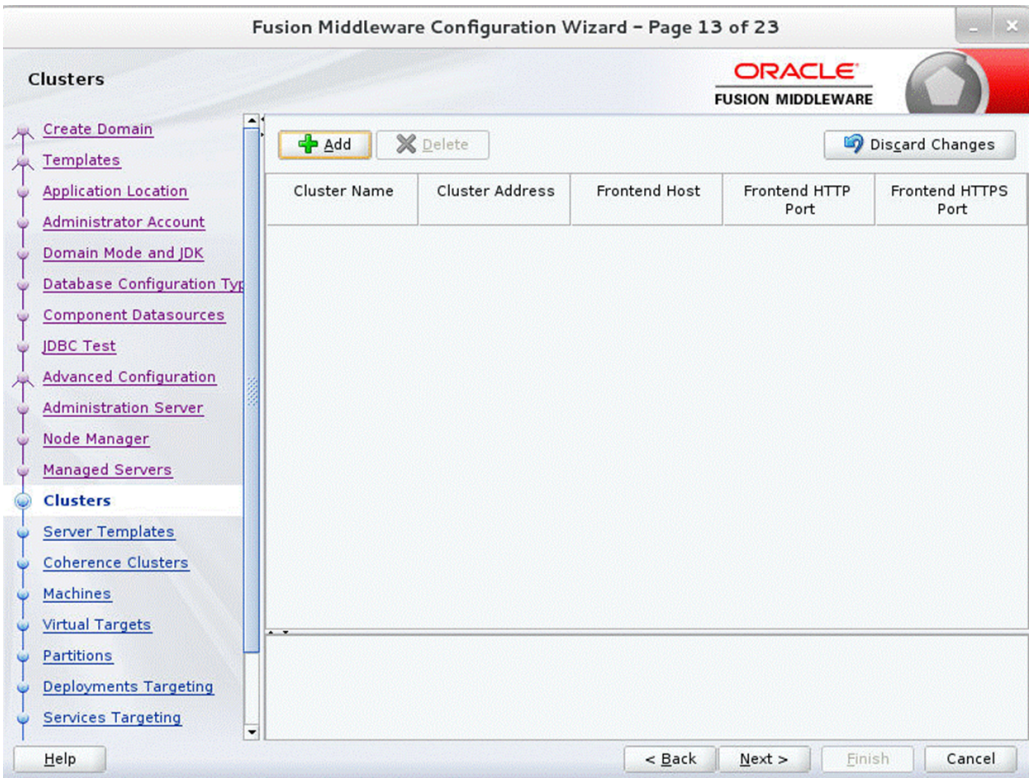
11. Click **Next**. The Node Manager window displays. Select the **Node Manager Type** and enter the **Node Manager** credentials.



12. Click **Next**. The Managed Servers window displays.
    a. Click **Add** to add a managed server on which you will deploy the application.
    b. Enter the **Server Name**, **Listen Address**, and **Listen Port** for the managed server.
    c. Set the Server Groups to `JRF-MAN-SVR`.

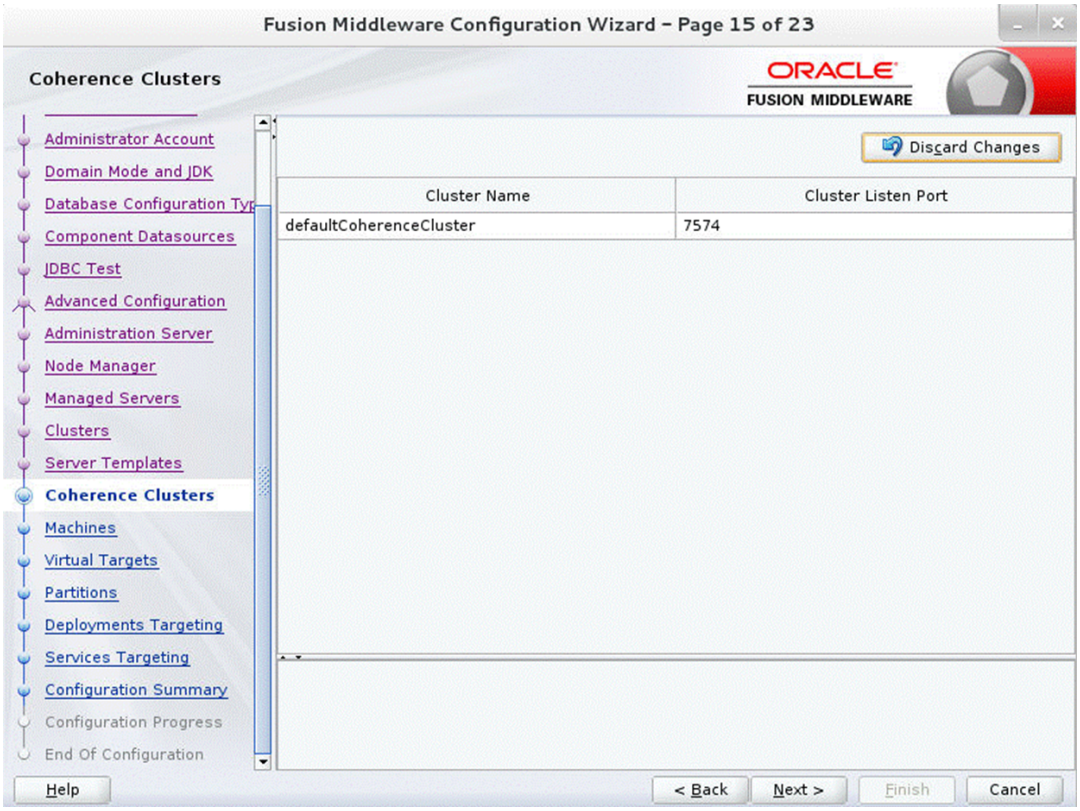


13. Click **Next**. The Clusters window displays.
    a. Click **Add** to add a cluster. This is an optional step in the procedure.



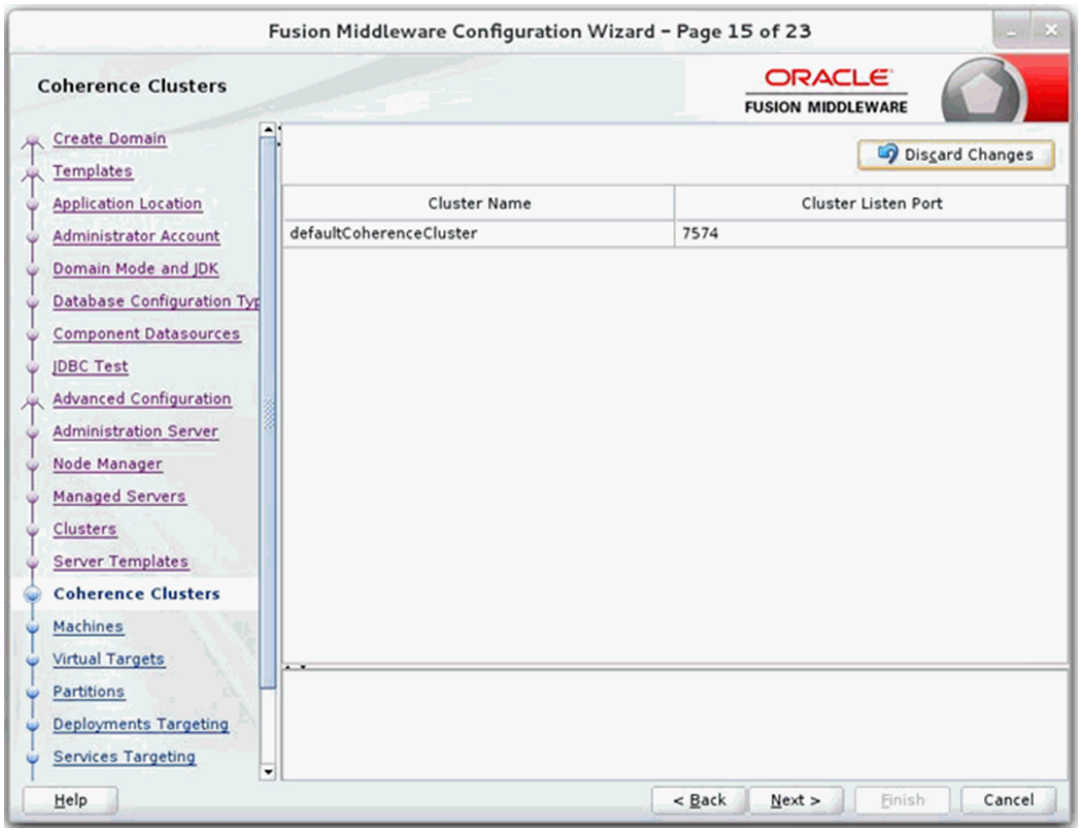
14. Click **Next**. The Server Templates window displays.

a. Click **Add** to add a server template. This is an optional step in the procedure.

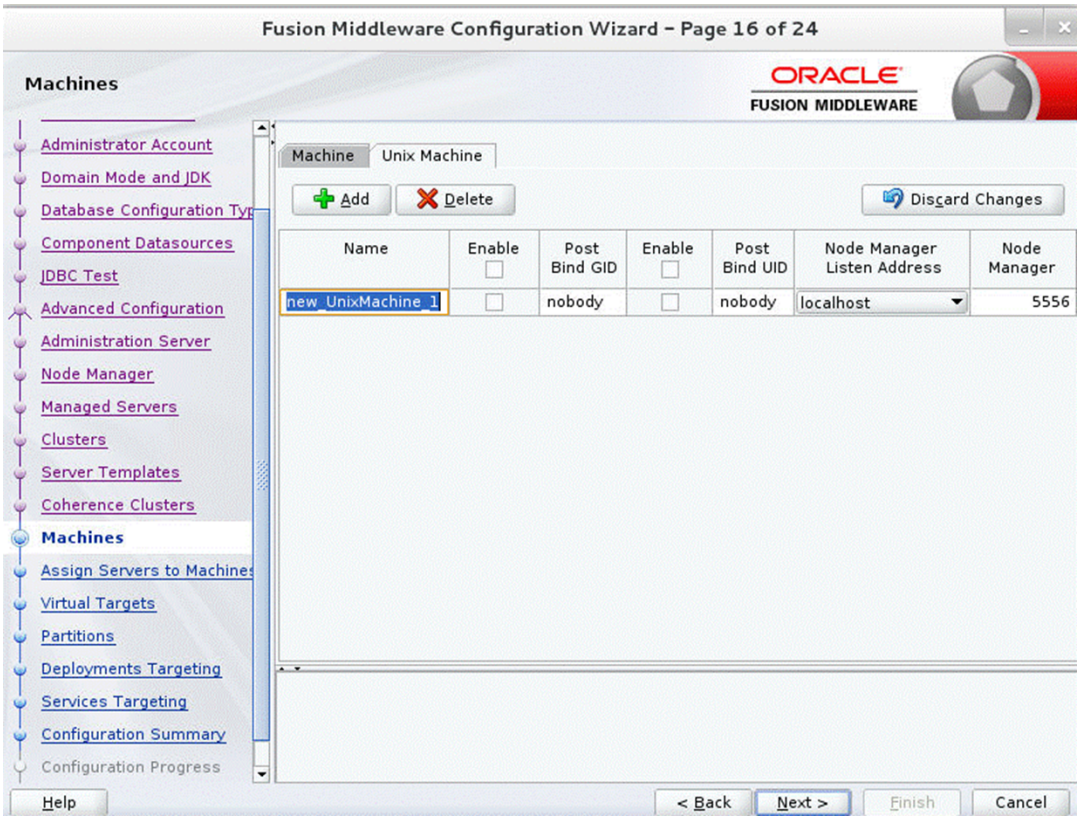


15. Click **Next**. The Coherence Clusters window displays.

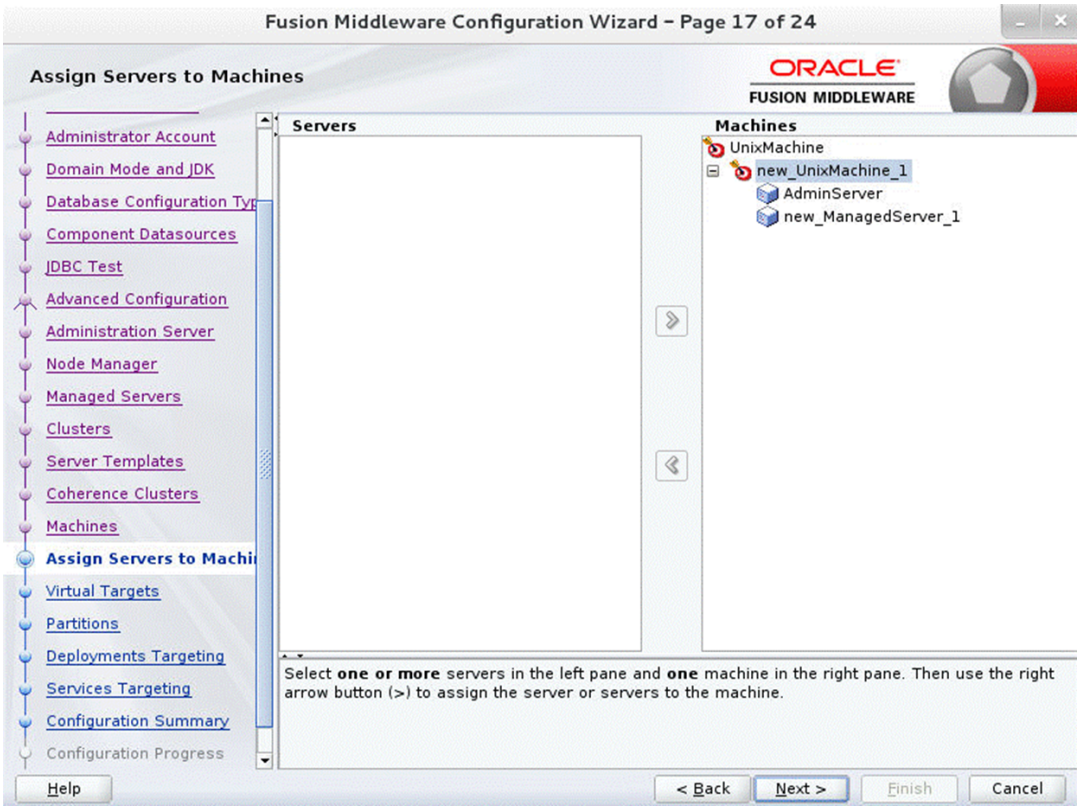
a. Add a coherence cluster. This is an optional step in the procedure.



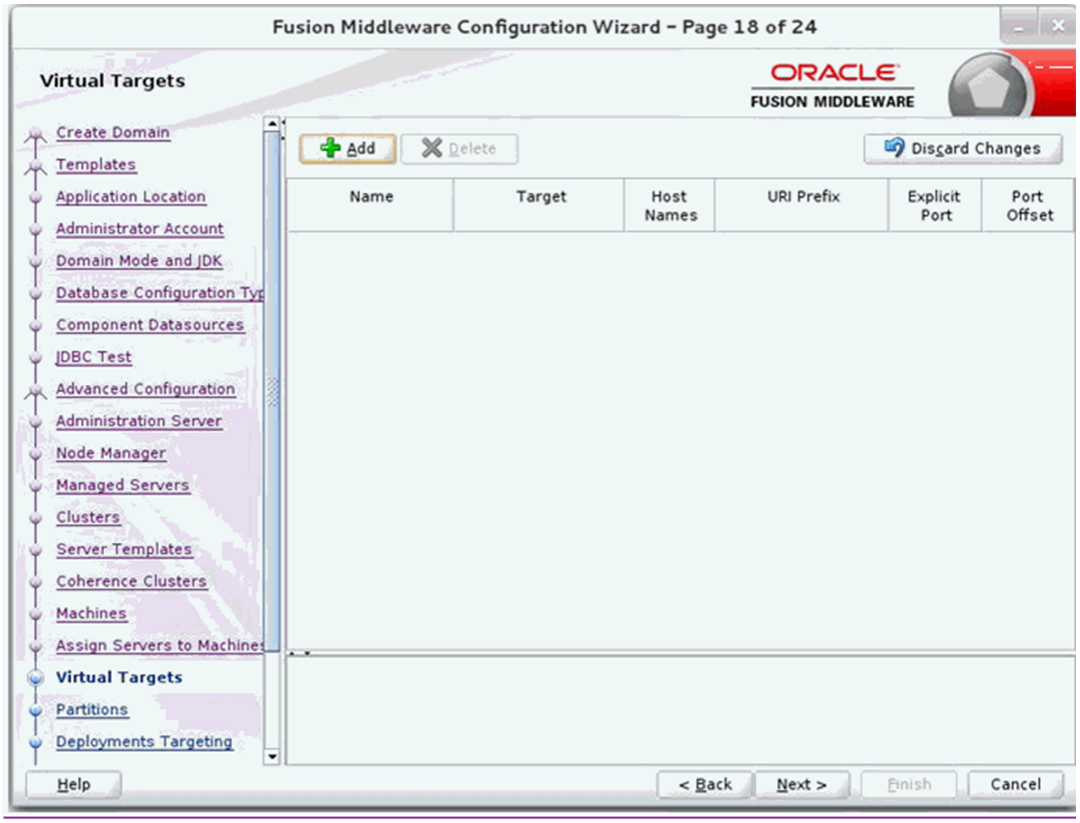
16. Click Next. The Machines window displays.
    a. Click Add.
    b. Enter the **Name** and the **Node Manager Listen Address** for the managed server.



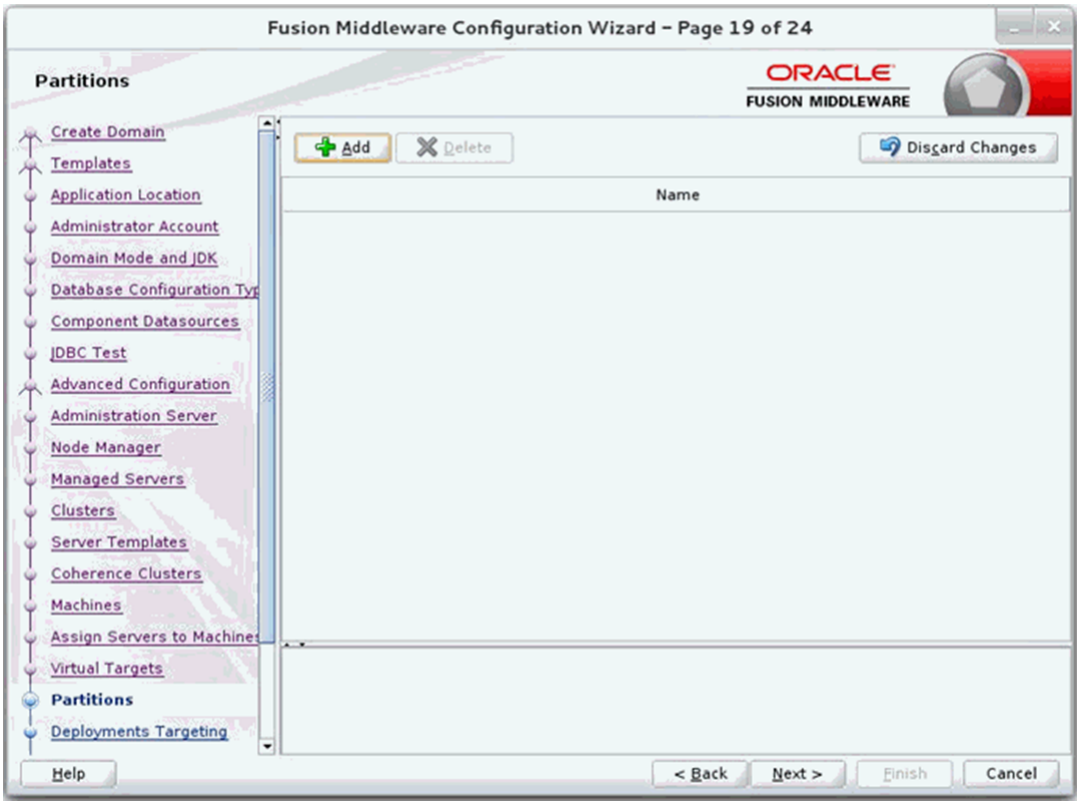
17. Click **Next**. The Assign Servers to Machines window displays. Add the Admin Server and the managed server to the computer.



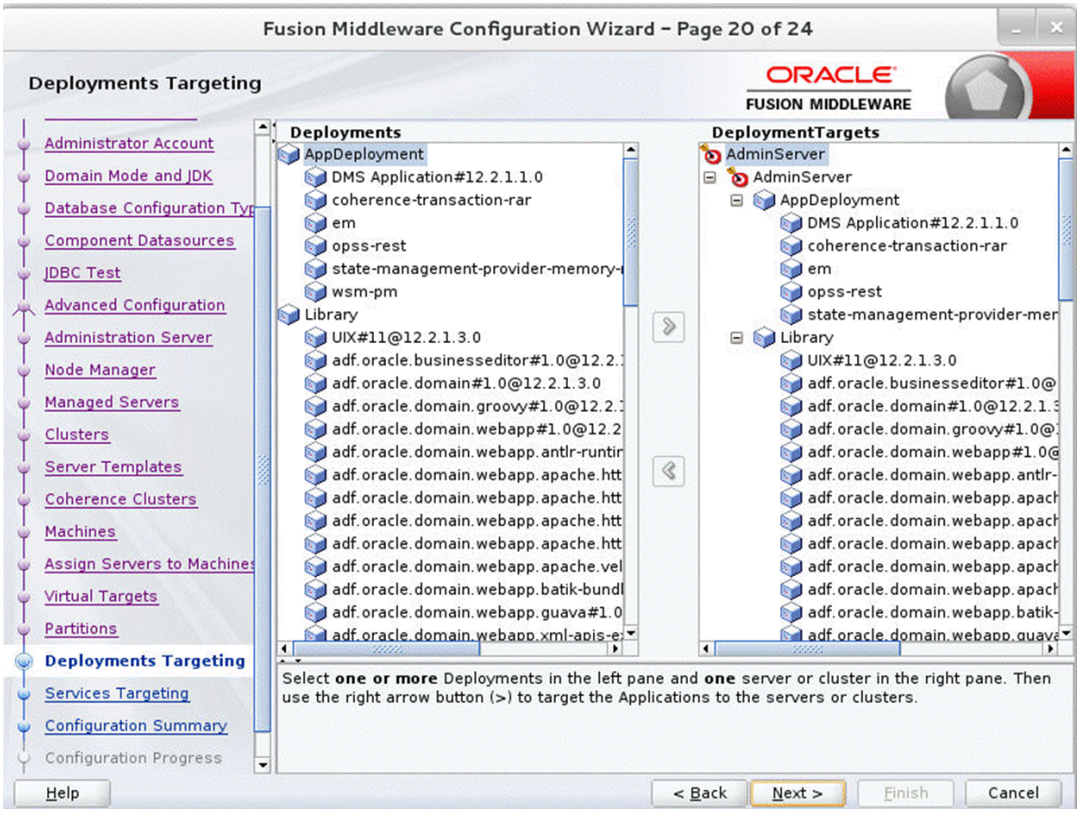
18. Click **Next**. The Virtual targets window displays.
    a. Click **Add** to add a Virtual target. This is an optional step in the procedure.



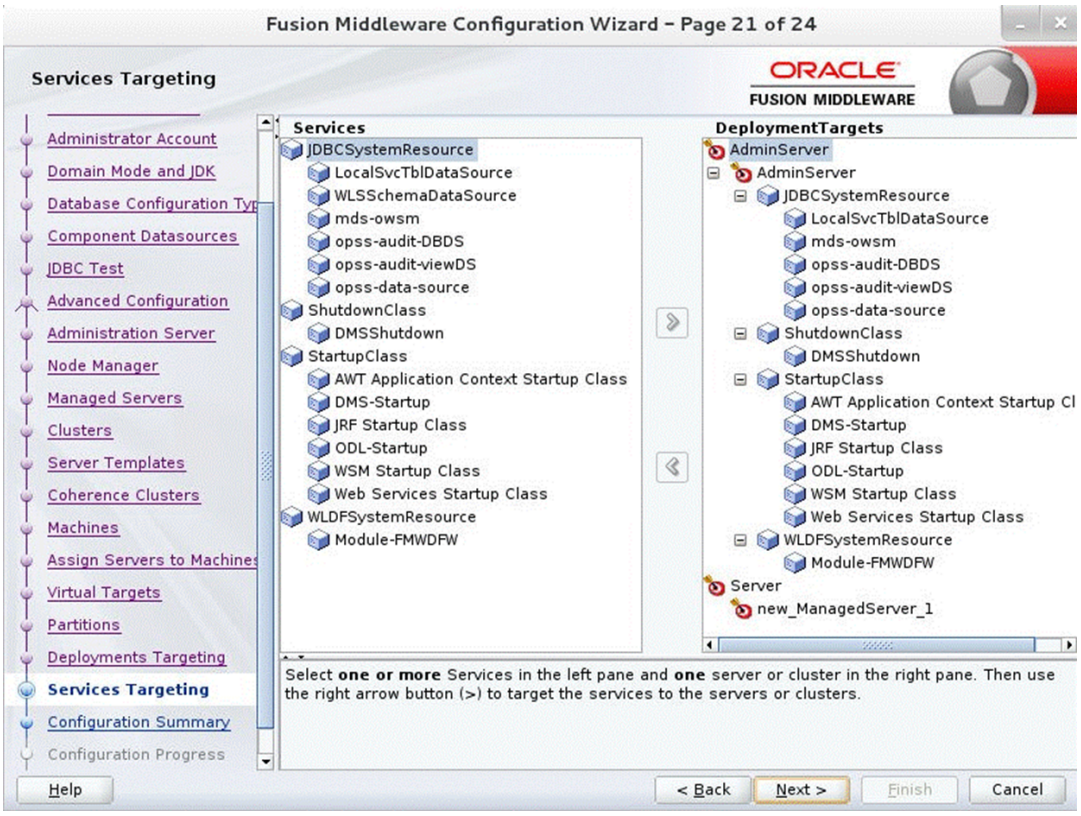
19. Click **Next**. The Partitions window displays.
    a. Click Add to add a Partition. This is an optional step in the procedure.



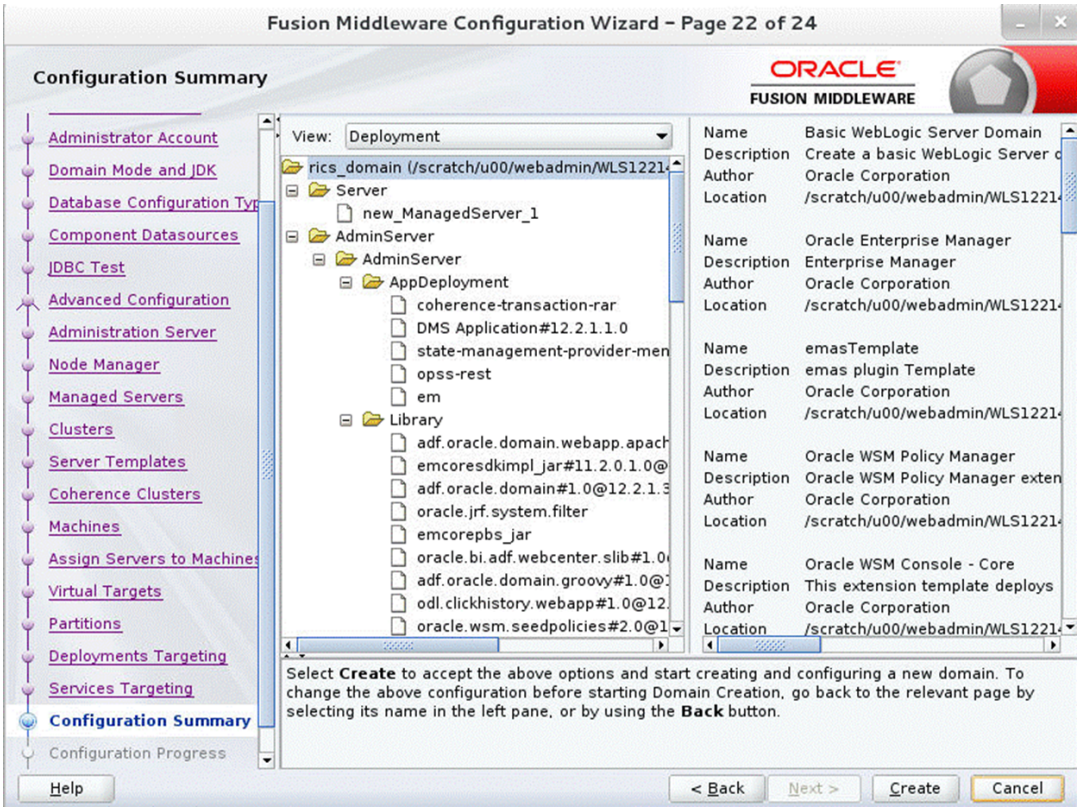
20. Click **Next**. The Deployments Targeting window displays. Select **wsm-pm** from **Deployments** and add it to **Admin Server** in **Targets**.



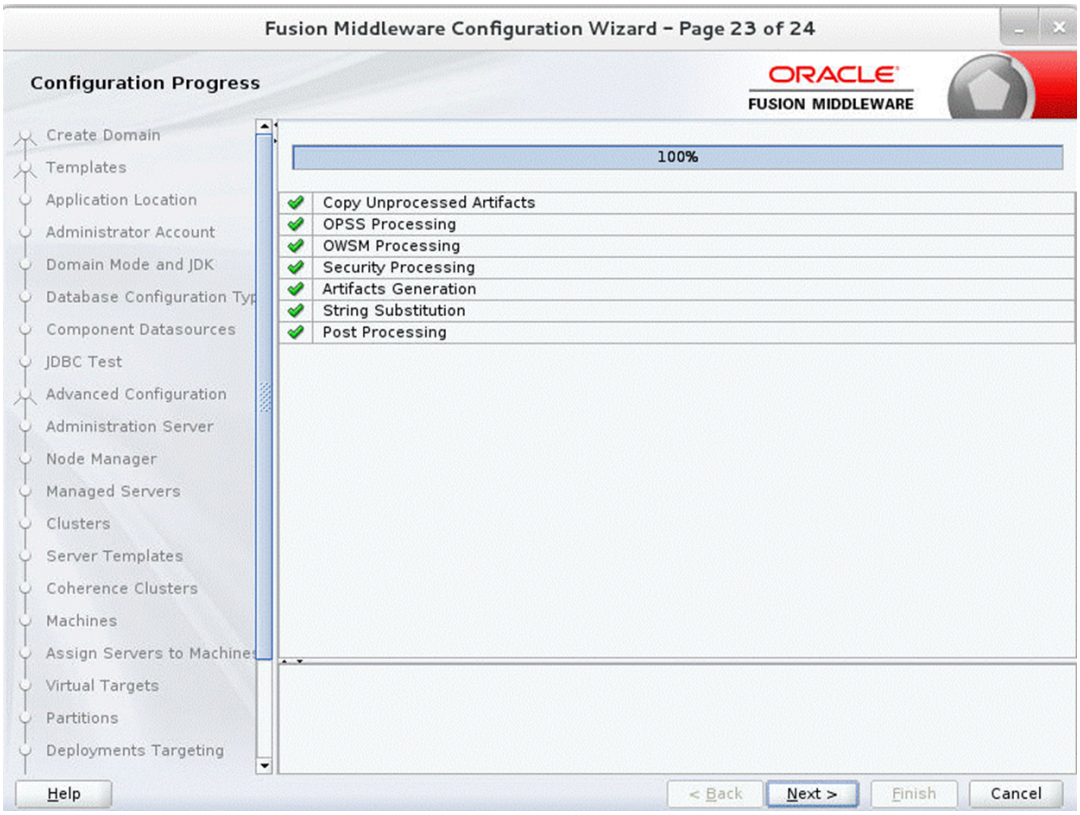
21. Click **Next**. The Services Targeting window displays.



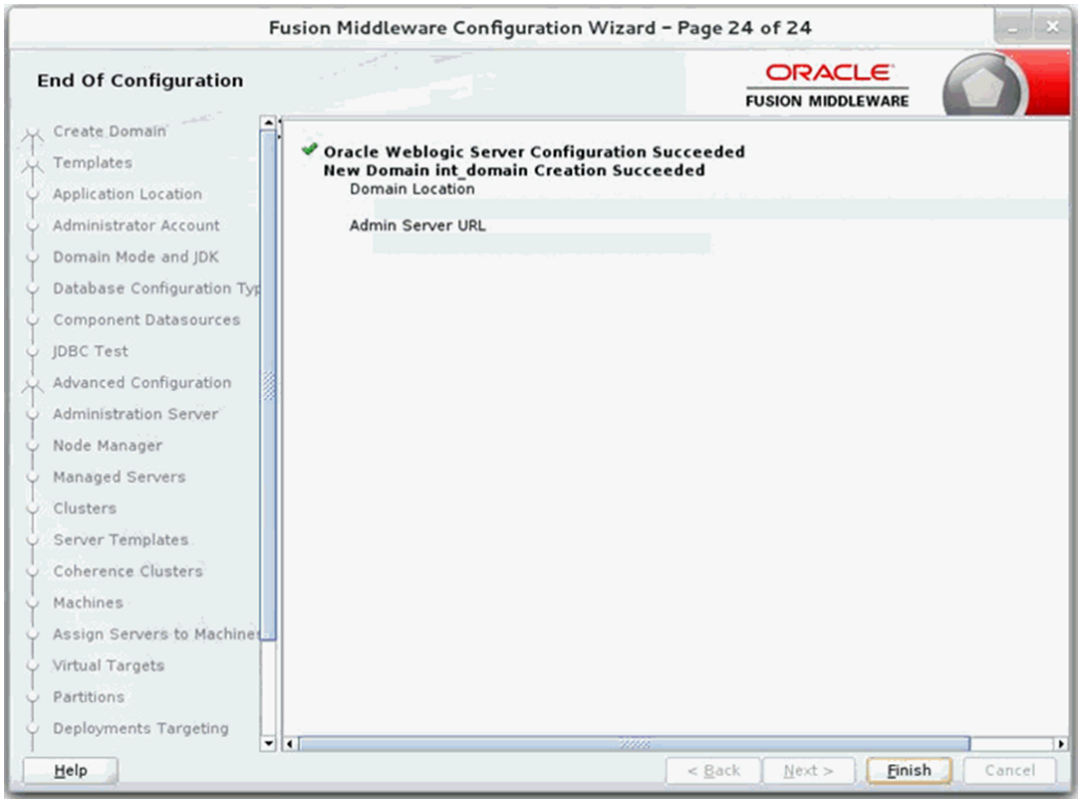
22. Click **Next**. The Configuration Summary window displays. Verify that all information described in this window is accurate.



23. Click **Create**. The Configuration Progress window displays a message when the domain is created successfully.



24. Click **Next**. The Configuration Success window displays that describes the Domain Location and Admin Server URL once the configuration is complete.



25. Click **Finish** to complete creating the WebLogic domain and managed servers.

Steps for ear Deployment

1. Client connector pak contains the latest v24 rib-rwms application distribution for on-prem installation. Download and extract the RIB kernel for RMWS-secondary-app `RibKernel24.0.000ForRwmsSecondary24.x.xApps_eng_ga.jar`.
2. Extract the contents of the jar file.
3. Open `rib-deployment-env-info.xml` found inside `./rib-rwms-secondary-home/deployment-home/conf`.
4. Edit this file to specify your deployment environment information.
   a. Make sure the following entries are present in the `<app-in-scope-for-integration>` section:

      ```
      <app id="rwms" type=" slave-plsql-app" />
      ```

   b. Update the `rib-jms-servers` section to provide the AQ JMS server details. Because the secondary app deploys on premise, it will not have access to AQ JMS on the cloud. Use RWMS app schema detail for AQ JMS setup. For example:

      ```
      <aq-jms-server jms-server-id="jms1">
        <jms-server-home>ribadmin@jms1host.example.com:/u00/oracle/product/11.2.0.2</jms-server-home>
        <jms-url>jdbc:oracle:thin:@rwmsappdbhost:1521/service_name</jms-url>
        <jms-port>1521</jms-port>
        <jms-user-alias>jms1_jms_user-name-alias</jms-user-alias>
      </aq-jms-server>
      ```

c. Update the RIB domain details in the `weblogic-application-servers` section.

d. Skip updating the `rib-func-artifact-server` details. `Rib-func-artifact` deployment is not required for secondary (on-prem) `rib-rwms`.

e. Update RIB-RWMS secondary server details. For example:

```
<wls id="rib-rwms-wls1">
   <wls-instance-name>rib-rwms-server</wls-instance-name>
   <wls-instance-home>webadmin@ribhost:/RIB/WLS/user_projects/domains/RIBDomain/servers/rib-rwms-onprem-server</wls-instance-home>
   <wls-listen-port protocol="http">19103</wls-listen-port>
   <wls-user-alias>rib-rwms_wls_user-name-alias</wls-user-alias>
</wls>
```

f. Make sure the datasource URL (host, port n service) entries are updated in the rib-app section of rib-rwms secondary.

```
<rib-app id="rib-rwms" type="slave-plsql-app">
        <deploy-in refid="rib-rwms-wls1" />
        <rib-admin-gui>
           <web-app-url>http://ribhost.example.com:19103/rib-rwms-appserver-gui/index.jsp</web-app-url>
           <web-app-user-alias>rib-rwms_rib-admin-gui_admin-user-name-alias</web-app-user-alias>
           <web-app-user-alias>rib-rwms_rib-admin-gui_operator-user-name-alias</web-app-user-alias>
           <web-app-user-alias>rib-rwms_rib-admin-gui_monitor-user-name-alias</web-app-user-alias>
        </rib-admin-gui>
        <error-hospital-database>
           <hosp-url>jdbc:oracle:thin:@rwmsappdbhost.example.com:1521/pdborcl</hosp-url>
           <hosp-user-alias>rib-rwms_error-hospital-database_user-name-alias</hosp-user-alias>
        </error-hospital-database>
        <app-database>
           <app-db-url>jdbc:oracle:thin:@rwmsappdbhost.example.com:1521/pdborcl</app-db-url>
           <app-db-user-alias>rib-rwms_app-database_user-name-alias</app-db-user-alias>
        </app-database>
        <notifications>
           <email>
              <email-server-host>mail.example.com</email-server-host>
              <email-server-port>25</email-server-port>
              <from-address>admin@example.com</from-address>
              <to-address-list>admin@example.com</to-address-list>
           </email>
           <jmx />
        </notifications>
        <app id="rwms" type="plsql-app">
           <jndi-not-applicable/>
        </app>
</rib-app>
```

> **Note:**
>
> As the secondary app deploys on-premise, it will not have access to AQ JMS and Error hospital. Therefore, all the datasources must connect to the RWMS app schema.

5. Compile: Run the `rib-home/application-assembly-home/bin/rib-app-compiler.sh` script with setup-security-credential from the `rib-home/application-assembly-home/bin` directory.

   Example:

   ```
   ./rib-app-compiler.sh -setup-security-credential
   ```

6. Deploy: Execute the `rib-home/deployment-home/bin/rib-app-deployer.sh` script with the appropriate command line parameter.

   ```
   rib-app-deployer.sh -deploy-rib-app-ear rib-<app>
   rib-func-artifact deployment is not required.
   ```

7. Verify: Once the rib-rwms secondary app is deployed, open the rib-admin-gui from a web browser using the credentials provided during compilation:

   ```
   <http or https://>host:port/rib-rwms-admin-gui
   ```

8. Make sure the Publication and Subscription WS are available to use.

   Example:

   ```
   https://ribhost.example.com:17010/
   RemotePlsqlPublisherComponentServiceBean/
   RemotePlsqlPublisherComponentServiceBeanService?WSDL
   https:// ribhost.example.com:17010/
   PlsqlApplicationMessageInjectorServiceBean/
   PlsqlApplicationMessageInjectorServiceBeanService?WSDL
   ```

4

RIB Self-Service Enablement

The Self-service enablement is a feature for provisioning RIB on cloud post deployment only. Because of the promising high availability feature of applications on the cloud environment, this is an essential feature that minimizes the redo of the RIB install cycle post configuration changes to any RIB-app.

The Self-service enablement allows below provisioning in rib-<app>:

Table 4-1 Self-Service Feature

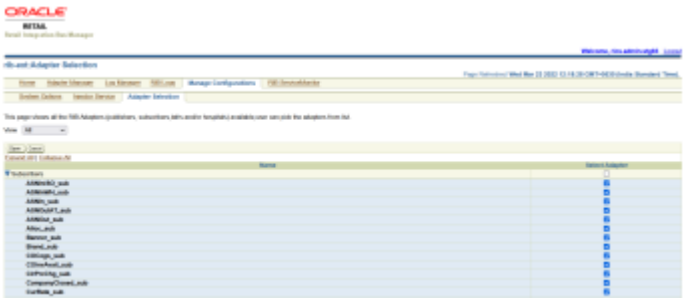
Self-Service Feature:	Self Service Feature on RIB-Admin GUI
Provisioning RIB adapters Choosing the subset of RIB adapters in scope for integration	

Table 4-1 (Cont.) Self-Service Feature

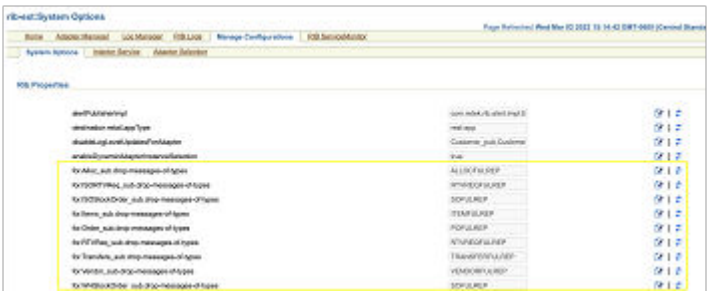
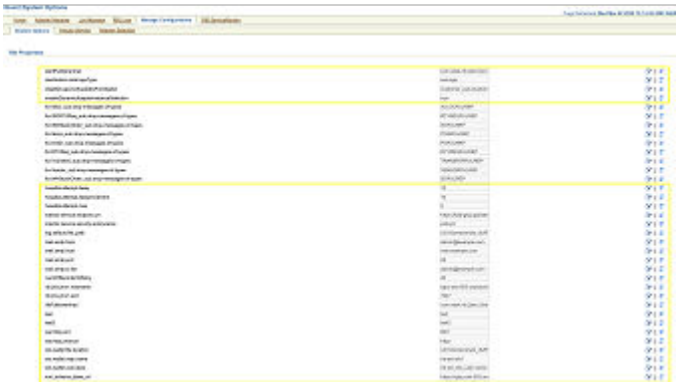
Self-Service Feature:	Self Service Feature on RIB-Admin GUI
Provisioning System Options Dynamically modifying configurations via, rib-<app> properties file. Example shown for rib-ext for dropping messages for specific types for subscriptions. Similarly, the drop messages types can be configured for other RIB applications like rib-sim, rib-rms and so on. **Note:** There are other infrastructure level options that are available only for AMS or devops teams to configure or update, as shown in the screenshot.	 

Table 4-1 (Cont.) Self-Service Feature

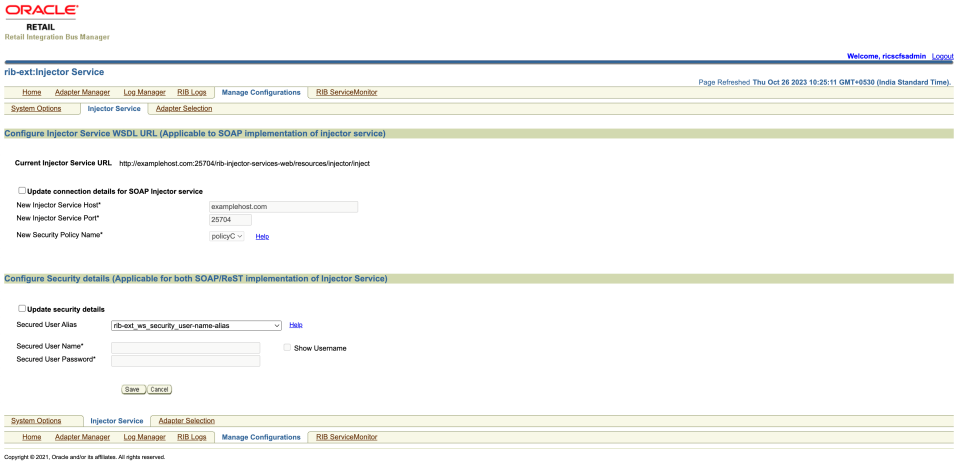
Self-Service Feature:	Self Service Feature on RIB-Admin GUI
Email notification fields are read only now which means email notification configuration can't be altered post provisioning. If customer wants to enable email notification or update configuration then this request has to come through proper channel and can be done from backend.	
Provisioning Injector Service URL Hook to alternate subscribing retail application installation. Injector service url can be updated only for customer owned apps like -rib-ext , rib-lgf	

Table 4-1 (Cont.) Self-Service Feature

Self-Service Feature:	Self Service Feature on RIB-Admin GUI
RIB ServiceMonitor Verify InjectorService provisioned in previous step.	

Provisioning RIB-Adapters

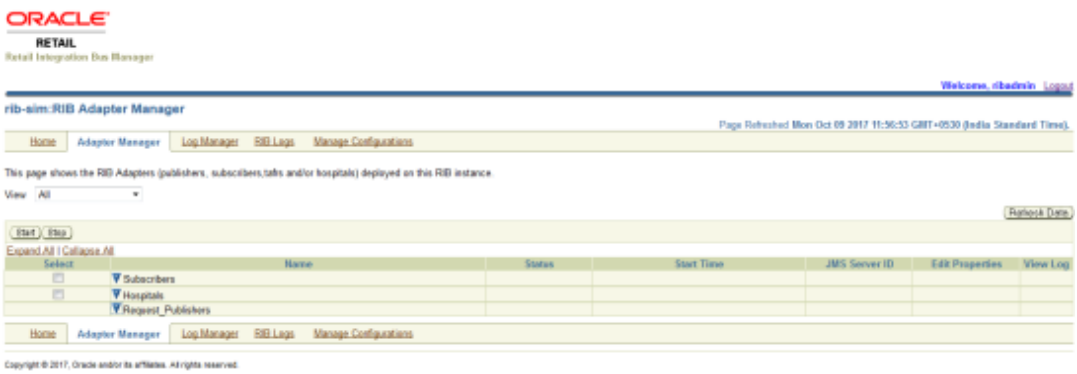
Every rib-<app> contains a set of publish and subscribing adapters for exchanging messages between retail applications. Subscribing adapters are MDB which are resource intensive. The higher the number of adapters in scope the higher is the resource crunch. In an environment which does not make use of all the publishing and subscribing adapters bundled with the rib-app, the user is allowed to choose a subset of the adapters needed based on the RIB functional flow. This configuration change takes effect dynamically and does not require a redeployment of the rib-<app>.

Follow the steps below for configuring the rib-<app> adapters in scope of the integration.

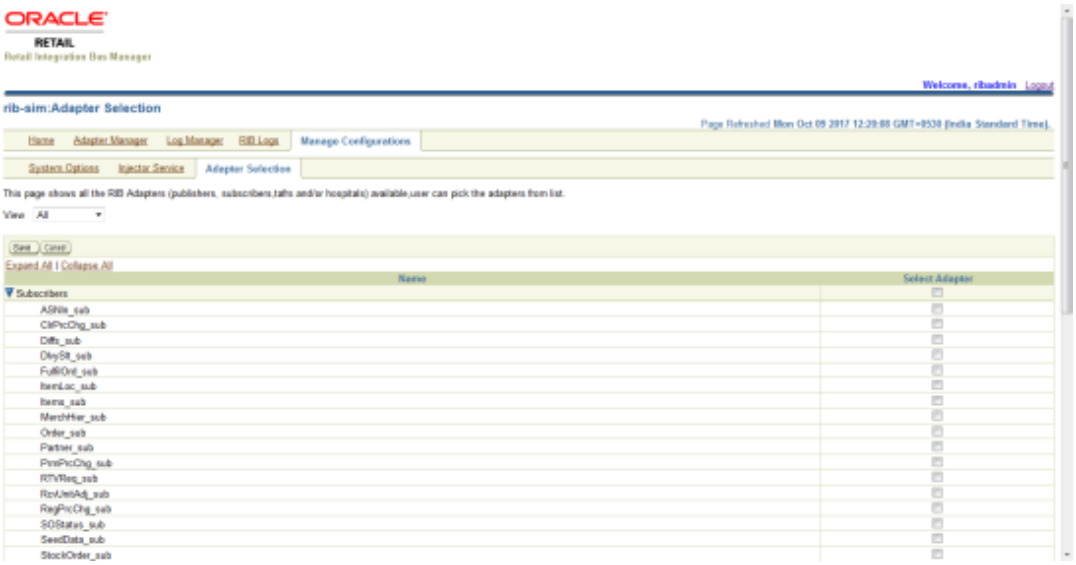
1. By default , dynamic adapter selection feature is enabled for rib-ext. `enableDynamicAdapterInstanceSelection` flag is applicable ONLY for RIB-EXT and this flag shouldn't be used in any other rib-apps.

   ```
   enableDynamicAdapterInstanceSelection=true
   ```

2. Only if the above property is set to true, the user can select the adapters dynamically. Below is the default landing page when RIB adapters added in scope.



3. In the RIB-Admin GUI, the Manage Configuration > Adapter Selection tab provides the list of all available adapters whose subset can be chosen to publish, subscribe and retry rib messages based on rib integration flows.



4. Select the subset of publishing, subscribing and retry adapters depending on the rib-integration-flow in consideration and click **Save**.

   Consider the below rib-integration flows:

   rib-sim publishing the ***InvReq*** message

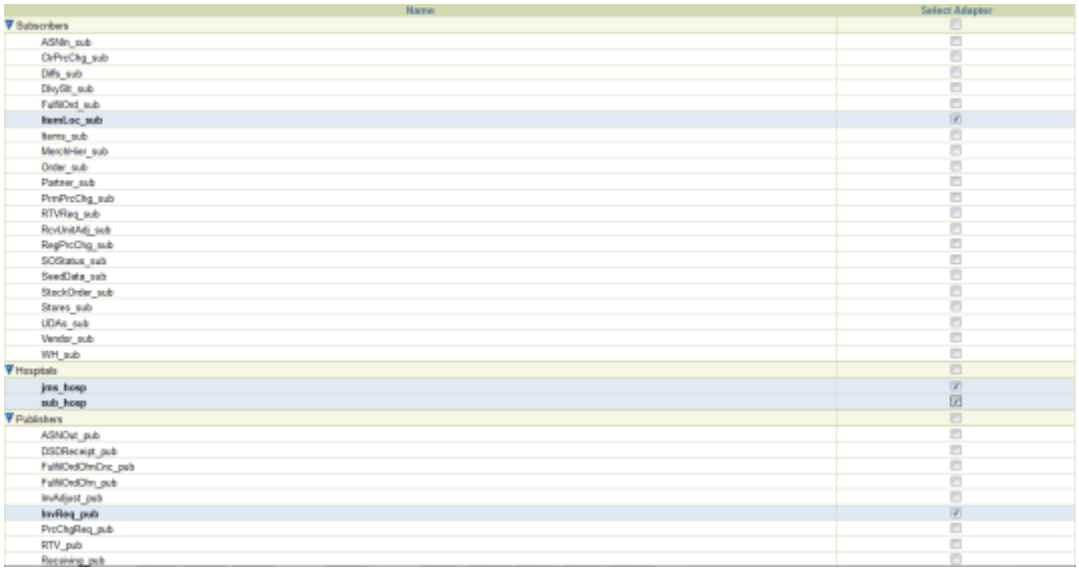
```
<message-flow id="31">
              <node id="rib-sim.InvReq_pub" app-name="rib-sim"
                    adapter-class-def="InvReq_pub" type="DbToJms">
                     <in-db>default</in-db>
                     <out-topic>etInvReq</out-topic>
              </node>
              <node id="rib-ext.InvReq_pub" app-name="rib-ext"
                    adapter-class-def="InvReq_pub" type="DbToJms">
                     <in-db>default</in-db>
                     <out-topic>etInvReq</out-topic>
              </node>
              <node id="rib-rms.InvReq_sub" app-name="rib-rms"
                    adapter-class-def="InvReq_sub" type="JmsToDb">
                     <in-topic>etInvReq</in-topic>
                     <out-db>default</out-db>
              </node>
              <node id="rib-ext.InvReq_sub" app-name="rib-ext"
                    adapter-class-def="InvReq_sub" type="JmsToDb">
                     <in-topic>etInvReq</in-topic>
                     <out-db>default</out-db>
              </node>
       </message-flow>
```

   rib-sim subscribing the ***ItemLoc*** message from RMS

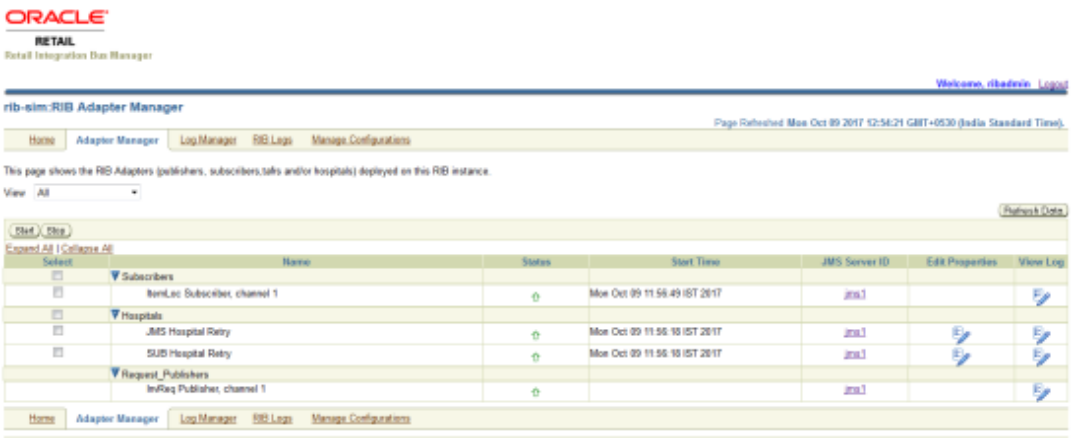
```
<message-flow id="6">
              <node id="rib-rms.ItemLoc_pub" app-name="rib-rms"
                    adapter-class-def="ItemLoc_pub" type="DbToJms">
                     <in-db>default</in-db>
                     <out-topic>etItemLocFromRMS</out-topic>
              </node>
```

```
        <node id="rib-ext.ItemLoc_pub" app-name="rib-ext"
              adapter-class-def="ItemLoc_pub" type="DbToJms">
               <in-db>default</in-db>
               <out-topic>etItemLocFromRMS</out-topic>
        </node>
        <node id="rib-sim.ItemLoc_sub" app-name="rib-sim"
              adapter-class-def="ItemLoc_sub" type="JmsToDb">
               <in-topic>etItemLocFromRMS</in-topic>
               <out-db>default</out-db>
        </node>
        <node id="rib-rwms.ItemLoc_sub" app-name="rib-rwms"
              adapter-class-def="ItemLoc_sub" type="JmsToDb">
               <in-topic>etItemLocFromRMS</in-topic>
               <out-db>default</out-db>
        </node>
        <node id="rib-ext.ItemLoc_sub" app-name="rib-ext"
              adapter-class-def="ItemLoc_sub" type="JmsToDb">
               <in-topic>etItemLocFromRMS</in-topic>
               <out-db>default</out-db>
        </node>
    </message-flow>
```

Considering the above flows, select **InvReq_Pub**, and **ItemLoc_sub** and both **Hospital** adapters as shown in the image below.



5. Verify that the selected adapters are reflected on the Adapter Manager tab. Newly added adapters in scope will be down. Newly added adapters in-scope need a start from the GUI to become ACTIVE as a one time activity; otherwise, newly added adapters won't show up on topic on checking from jms-console and won't even be registered (messages will be lost). Sometimes you need to start adapters 2-3 times because of one known issue where the subscriber registration process is times out. Post start of newly added adapters, ensure adapters are showing up on topic on checking from jms-console. If newly added adapters are not showing up on topic, please try to start them again from the UI. jms-console will not show adapters on topic immediately and there is expected 3 to 5 mins of delay.



6. All the adapters are in scope by default for rib-<app>:

   ```
   enableDynamicAdapterInstanceSelection = false
   ```

   This is the default value for all rib-<app>s except rib-ext for rib-ext following flag is set to true

   ```
   enableDynamicAdapterInstanceSelection = true
   ```

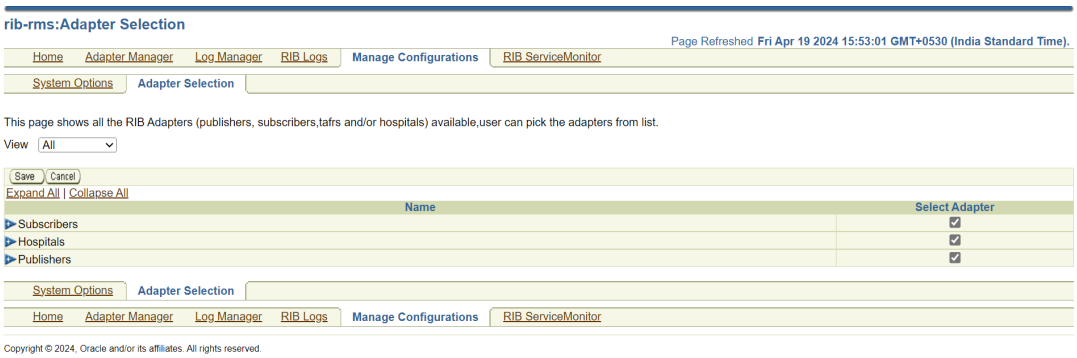
> **Note:**
>
> `enableDynamicAdapterInstanceSelection` flag is not available for end user update. Follow the steps in the next section to disable this flag for other `rib-<apps>` in case they are enabled.

How to Remove Dynamic Adapters Selection in RIB-RMS

The concept of Dynamic Adapters Selection applies only to the RIB-EXT application and all other RIB-<apps> such as RIB-RMS, RIB-SIM, RIB-TAFR etc do not support the dynamic adapters. Due to our documentation defect, which has been fixed now, some of our customers have used this feature in non RIB-EXT apps, especially in RIB-RMS, which is unsupported and can cause major issues such as messages piling up on JMS topics and slow down the entire system. customers should remove dynamic selection of the adapters in any rib-app they might have configured it in ex: RIB-RMS/RIB-TAF/RIB-SIM

Steps :

1. Log into the RIB-RMS Admin GUI.
2. Go to the Adapter Manager page and capture the list of adapters present on the page.
3. Go to **Manage Configurations -> Adapter Selection** and select all the adapters.
4. Click **Save**. Make sure all the adapters are displayed in the Adapter Manager page.



5. Go to **Manage Configurations -> System Options** and set **enableDynamicAdapterInstanceSelection** to **false**.

6. Go to the Adapter Manager page and bring down all the adapters that do not belong to the list collected in step 2.

Provisioning System Options

Application specific properties for the rib-<app> are configured in the rib.properties file. When RIB is deployed on cloud, the application specific properties can be configured in the RIB-Admin GUI application. The Manage Configuration > System Options tab allows the user to edit the properties values post deployment. There are some infrastructure level options that are available only for AMS or devops teams to configure or update.

Following are the frequently configurable RIB properties:

1. Drop-messages-of-types- for dropping messages for specific types for subscriptions.

2. Updating facility_id and facility_type for rib-tafr.

3. A new system option can also be added using 'Add' functionality in UI. Perform the following steps to add the Facilities for rib-tafr.
   - Click the **Add** button.

   - Insert a new **Facility ID**.

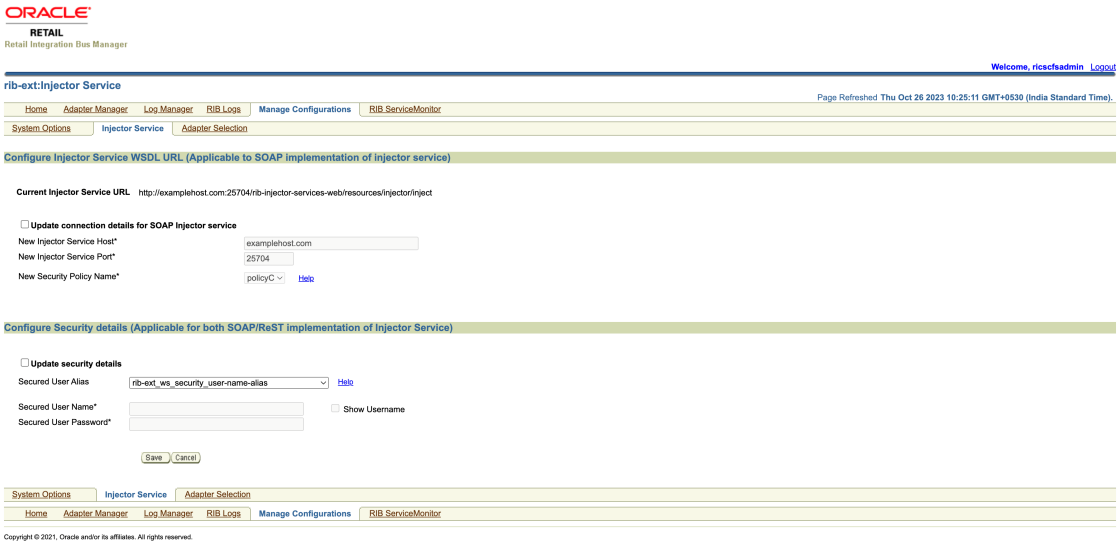
     For example: **key - facility_id.PROD.12345 value - 1**

4. Updating injector service url and policy for rest-app.

5. Updating IDCS host URL. This is needed only for customer owned applications using Oauth for rest call.

Provisioning InjectorService URL

In the RIB-Admin GUI, the Manage Configuration > Injector Service page allows the user to configure an injector service URL for a customer-owned applications.



Update injector service URL details by providing new host and port details and the user credentials for the service.

RIB ServiceMonitor

Once the RIB integration environment is configured for use by various retail application, as a sanity test the user may need to verify the integration end points. For RIB on cloud, we can ping-test various webservices consumed by RIB using RIB admin GUI.

In RIB Admin GUI, the RibServiceMonitor page lists all the webservices consumed by the rib-application and allows the user to ping the same. The webservices are pingable only if the "ping" operation is supported by the webservice.

5
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 "Performance Considerations" in the *Oracle Retail Integration Bus Operations Guide*.

> **Note:**
>
> For more information, see “Performance Considerations," in the *Oracle Retail Integration Bus Operations Guide*.

Performance and Parallel Logical Channels

The RIB must provide guaranteed once and only once processing of business events (messages) across the enterprise. Maintaining the order of business events across the enterprise is critical to data integrity.

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, which enables RIB to guarantee message sequencing.

Processing messages in sequence results in operational overhead, as every message must be checked against the database to find the status of previous messages on which it is dependent (same businessObjectid). Sequencing creates an inherent bottleneck, in that only one message is processed at once. For example, messages can come at the rate of 100 messages per second, but a RIB subscribing adapter can process only one of those messages at a time to preserve the order. To get around this bottleneck and improve performance, RIB provides options for optimization and functionality.

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.

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

Generally, integration for the retail application APIs are the biggest factor for bottlenecks in the overall messaging system throughput. It is not appropriate to start creating parallel logical channels at the first sign of performance problem. It is important to analyze and tune the integration APIs of the retail applications before considering the use of parallel channels.

Using parallel logical channels increases complexity, CPU demands, and memory requirement, resulting in more operational overhead. Use them only when, after all other components are fully tuned, you are still not able to meet your target numbers.

6
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. RIB System Administrators are involved in installing, configuring, deploying defect fixes, and making sure that the integration infrastructure is up and running properly. They generally are concerned with the business side of the integration system. Their tasks include bringing up or taking down RIB adapters, and fixing data issues with message payloads using RIHA. There are separate realms, roles, groups, and users defined for each category of RIB administrators.

RIB Application Administrators Security Domain

For each rib-<app>.ear deployed, RIB creates the users belonging to the below groups:

- RicsAdminGroup
- RicsOperatorGroup
- RicsMonitorGroup

The default groups and user that RIB creates must not be deleted or modified.

RIB follows a role-based authorization for allowing valid users to perform a defined set of operations from the rib-admin-gui. The user belonging to each of above groups will be associated with a well defined role and thus able to perform authorized operations only. It is recommended that you have a unique user belonging to each group.

Integration with SIOCS

1. RIB will use IDCS OAuth2 for authentication of ReST calls both inbound and outbound (publisher/injector restful services). The primary authentication mechanism in the cloud is OAuth2 using the IDCS authenticator. Out-of-the-box configuration expects OAuth2 to be used.
2. RICS to EICS integration will be a ReST call with OAuth2.
3. The EICS injector URL will be auto-wired as part of RICS provisioning. URL will look something like:

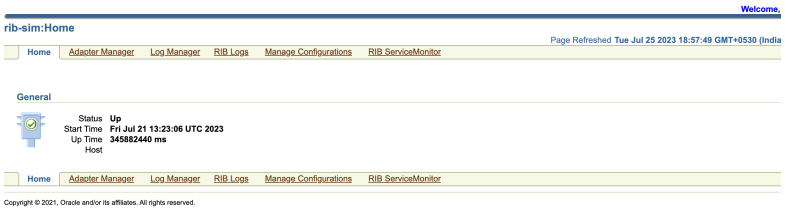
   ```
   http://wtss-svc.<SIOCS_SUB-NAMESPACE>.svc.occloud:9999/siocs-int-services/
   internal/api/inject
   ```

4. The RICS IDCS Client ID and Secret are auto-wired with rib-sim_oauth2_application_client_user-name-alias as part of provisioning. These will be used to get the access token for accessing EICS end point.

> **Note:**
>
> rib-sim_ws_security_user-name_alias is for BasicAuth and should be set empty for OAuth2 however auto wiring takes care of setting this alias to empty.

5. IDCS Url is also auto-wired, and is set during RICS provisioning. The URL looks something like:

```
https://idcs-<TENANT>/oauth2/v1/token
```

Step	Comment
Access rib-sim admin GUI at `https://<external-load-balancer>/<sub-namespace>/rib-sim-admin-gui`	
Navigate to Manage Configurations-> System Options. Search and verify the following system options: **a.** injector.service.app Type : rest-app **b.** Check the injector.service.end point.url. URL should be something like: `http://wtss-svc.<SIOCS_SUB-NAMESPACE>.svc.occloud:9999/siocs-int-services/api/ribinjector/inject` **c.** Look for injector.service.security.policyname, policy should be policyC for internal calls. **d.** oauth2.default.authorizationServerUrl : RICS IDCS Host for making call to get the access to-ken.	

Step	Comment
Navigate to Manage Configurations-> Injector Service. Verify the following: **a.** Current Injector Service URL : should point to correct injector service url. **b.** rib-sim_ws_security_username_alias credential should be empty. **c.** rib-sim_oauth2_application_client_user-name-alias credential must be getting populated with client ID and secret.	
How to verify whether the SIM injector URL and credentials are correct. Navigate to RIB Service Monitor Tab **a.** Click **ping** to test the connectivity.	

Integration with ROB

1. RICS to ROB integration is Rest call, Oauth2 Authorization.
2. The integration is configured between ROB and RICS via the ReST service (which is HTTPS).
3. ROB injector URL looks something like this:

   ```
   https://<external-load-balancer>/<rob-sub-namespace>/rib-injector-services-web/orcos/resources/injector/inject
   ```

4. The OB IDCS app Client ID and Secret will be used to get the access token for accessing ROB end point.
5. The IDCS Url is set during RICS provisioning. The URL looks something like:

   ```
   https://idcs-<TENANT>/oauth2/v1/token
   ```

> **Note:**
>
> rib-rob_ws_security_user-name_alias is for BasicAuth and should be set empty for OAuth2

Table 6-1 Integrating with ROB

Category	Steps	Comment
Access RIB Admin GUI	Access the rib admin GUI at https://<external-load-balancer>/rib-rob-admin-gui Log in with the admin user.	
Verify Configuration and update	Navigate to Manage Configurations -> System options 1. Search for and verify the following: destination.retail.appType: rest-app 2. Check the value for InjectorService URL (injector.service.endpoint.url).URL should look something like this: https://omni.retail.us-phoenix-1.ocs.oc-test.com/rgbu-omni-rgbu-stg83-obcs/rib-injector-services-web/orcos/resources/injector/inject 3. Security Policy (injector.service.security.policyname): policyA 4. IDCS OAuth Server URL (oauth2.default.authorizationServerUrl): https://<idcs-tenant>/oauth2/v1/token	

Table 6-1 (Cont.) Integrating with ROB

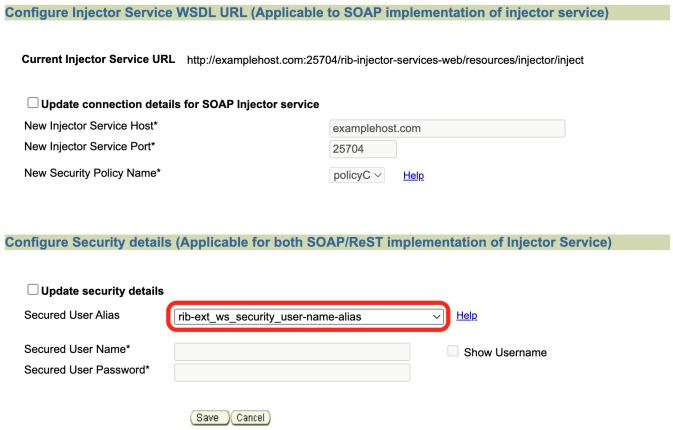
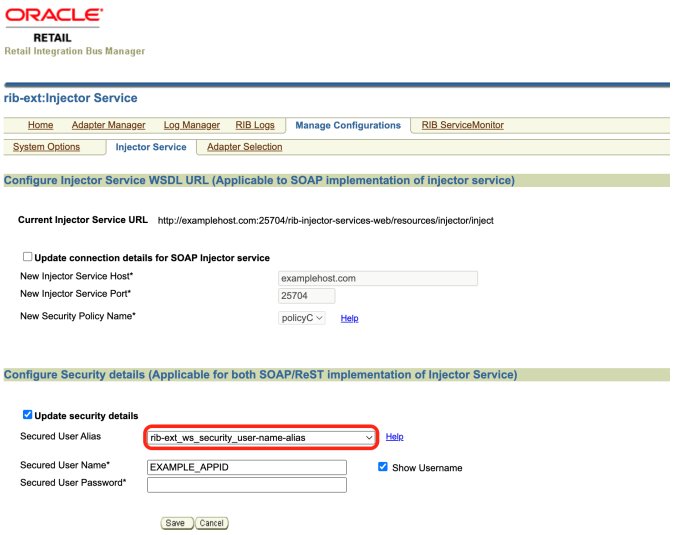
Category	Steps	Comment
Verify username and password	Navigate to Manage Configurations - > Injector Service 1. Choose rib-rob_ws_security_user_name_alias from drop down. 2. Set username and password to be empty.	
Verify ClientID and Secret	Navigate to Manage Configurations - > Injector Service Choose rib-rob_oauth2_application_client_user-name-alias from drop down and verify details 1. Verify a valid Client ID in username is set. 2. Verify a valid Client Secret in password is set.	
Ping test	Navigate to Manage Configurations -> RIB Service Monitor 1. Click on ping 2. It should return success	

Table 6-1 (Cont.) Integrating with ROB

Category	Steps	Comment

Verify provided credentials	How to verify if the ROB injector URL and credentials are correct.	1. Get the ROB Client ID and secret.
		2. Execute the following curl commands for grant_type client_credentials:
		`ClientId=RGBU_RICS_STG83_APPID`
		`ClientSecret=776381f5-88f5-4995-aa57-ecc7b7a1a8d7`
		`IDCSUrl=https://idcs-24e4baae56764e91be371e6a2060d66e.identity.c9dev2.oc9qadev.com`
		`AccessToken=$(curl -i -X POST \ --user $ClientId:$ClientSecret \ -H "Content-Type: application/x-www-form-urlencoded;charset=UTF-8" \ $IDCSUrl/oauth2/v1/token \ -d "grant_type=client_credentials&scope=urn:opc:idm:__myscopes__" | grep -o -P '(?<=access_token":").*(?=","token_type)')`
		`echo $AccessToken`
		`ribExtServiceUrl=https://omni.retail.us-phoenix-1.ocs.oc-test.com/rgbu-omni-rgbu-stg83-obcs/rib-injector-services-web/orcos/resources/injector/ping`
		`curl -ivkL --noproxy '*' -H "Authorization: Bearer $AccessToken" -H "Content-Type: application/xml" -X GET $ribExtServiceUrl`
		if you get a 200 response, then the configuration is correct
		if you get 401 unauthorized, then Client ID and secret are incorrect

7

Integration with Fusion Middleware

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

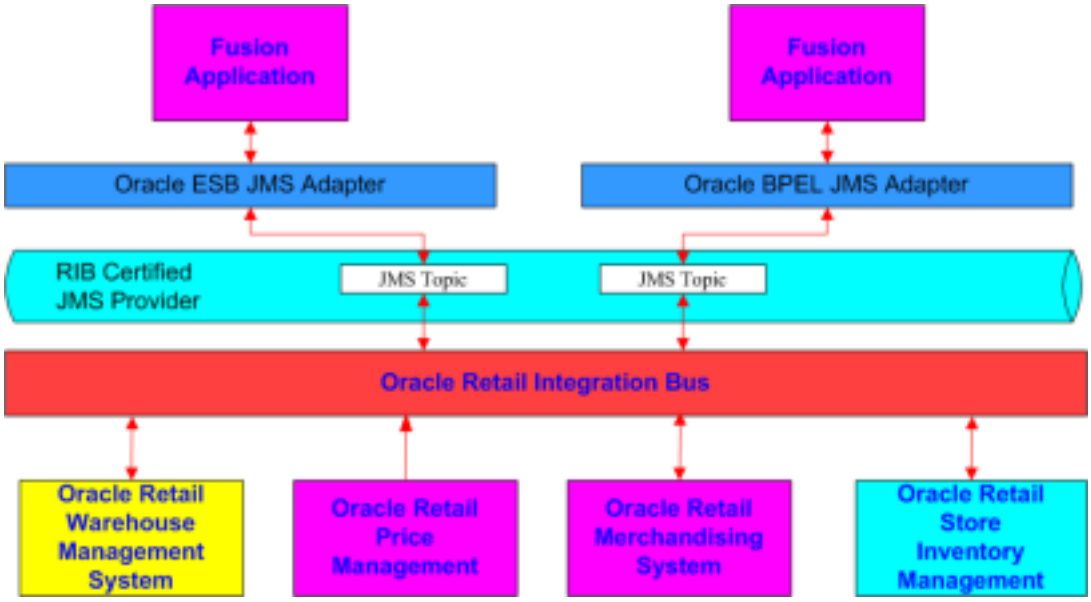
All RIB message payloads are fully standard compliant XSD based. All of the XML payloads are namespace aware and follow the general standards as well as the conventions that make them compatible with other Oracle Fusion products such ESB and BPEL. The payload schema definitions (XSDs) are packaged with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

The 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. It must have the ability to do the following:

- Connect to a standard JMS and publish to a topic.
- Create a durable subscriber to a RIB JMS topic
- Set user-defined message properties.
- 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 Bus 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.

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 a part of the functional artifact war file.

8

Integration with External Applications

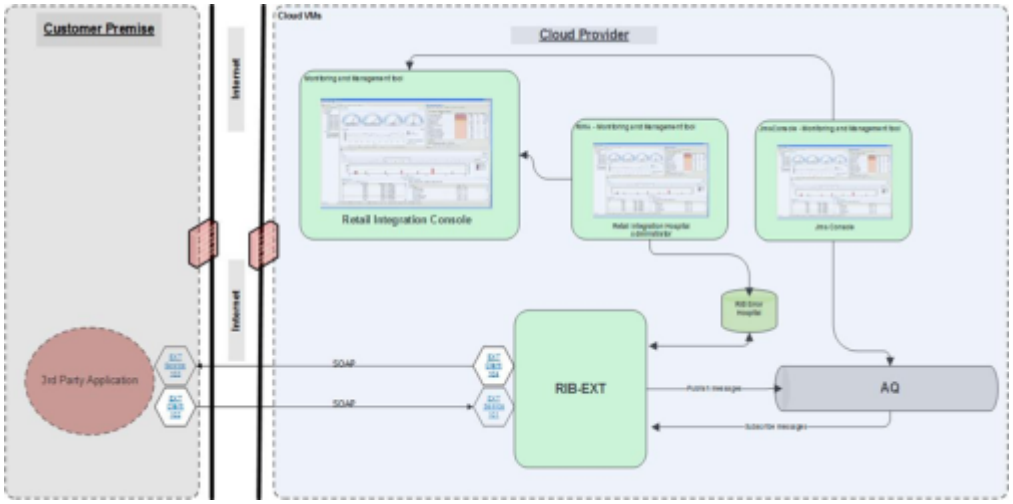
RIBforExt is the Oracle Retail Enterprise Integration component designed to address the connectivity requirements for 3rd Party integrations in a hybrid cloud topology where the RIB is deployed in the Retail Integration Cloud Services.

In a hybrid cloud scenario customers no longer have access to RIB's JMS server and cannot directly publish and subscribe to messages on the JMS topics. The RIB-EXT app is designed to fill that gap, it provides Web Service based APIs to publish to and subscribe from the RIB's JMS from third party systems.

RIBforEXT has all of the RIB flows available for the deployment time configuration based on the customer use cases.

RIB-EXT out-of-the-box provides the complete set of publishers, subscribers and retry adapters needed for the external application to integrate with Oracle Retail applications using RIB infrastructure.

The selective list of publisher and subscriber adapters needed by each specific external application is defined by the customer's implementation team.



Implementing RIB-EXT

RIB-EXT is an Oracle Retail Integration Application that provides necessary communication channel for external applications to publish and consume message from RIB's JMS on cloud and premise.

> **Note:**
>
> BasicAuth will no longer be supported starting from the RICS v24 release. RICS will enforce OAuth2 as the required authentication mechanism using the IDCS authenticator. Oauth2 is being enforced for authorization for ongoing security reasons and to ensure the customer stays within their OCI IAM limits. Customer/SI partner are advised to prepare for this change and implement OAuth2 for both inbound and outbound calls via RIB-EXT.
>
> RICS is also enforcing environment specific Oauth scope for authorization of inbound web service calls (RIB-EXT). The scope pattern that is used in the RICS IDCS app creation template is rgbu:rics:RICS-<ENVIRONMENT> where ENVIRONMENT is the environment type (STG, PRD, UAT, DEV1, DEV2, and so on). For details Refer Section: Create OAuth2 Client Application in IDCS.

How to Send/Receive Messages to/from the RIB System

For third-party integration, RIB-EXT provides ReST API's for external applications to send and receive data from the RIB system. The following sections cover the implementation details.

External Application as a Publisher (rest-app) using OAuth2

The end point of publishing service follows below pattern:

Table 8-1 Publishing Service Pattern

Resource	HTTP Method	Endpoint
Ping	GET	GET https://<external_LB_url>/<rics-sub-namespace>/rib-ext-services-web/resources/publisher/ping
Publish	POST	https://<external_LB_url>/<rics-sub-namespace>/rib-ext-services-web/resources/publisher/publish

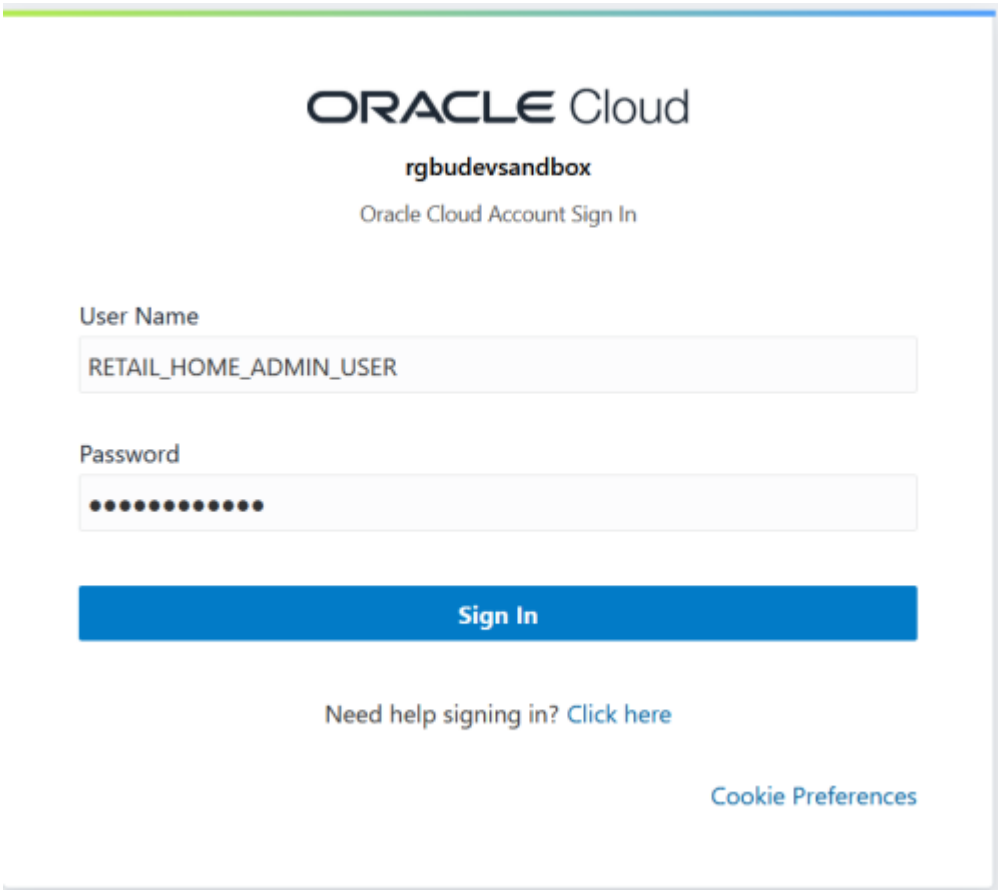
- RIB-EXT publishing service REST endpoints are protected using OAuth2 token-based authentication meaning end points are accessible by sending along an access token.
- Scope will be used for authorization of REST services. Scope for RICS is in the following format- rgbu:rics:RICS-<Environemnt Type><Environment Index> (that is, rgbu:rics:RICS-DEV1).
- Client Credentials grant type is supported.

For getting access to RICS publishing service you need to create a client app in IDCS. IDCS app generates an access token that will be used for making publishing service calls. Follow steps for creating the client app in IDCS.

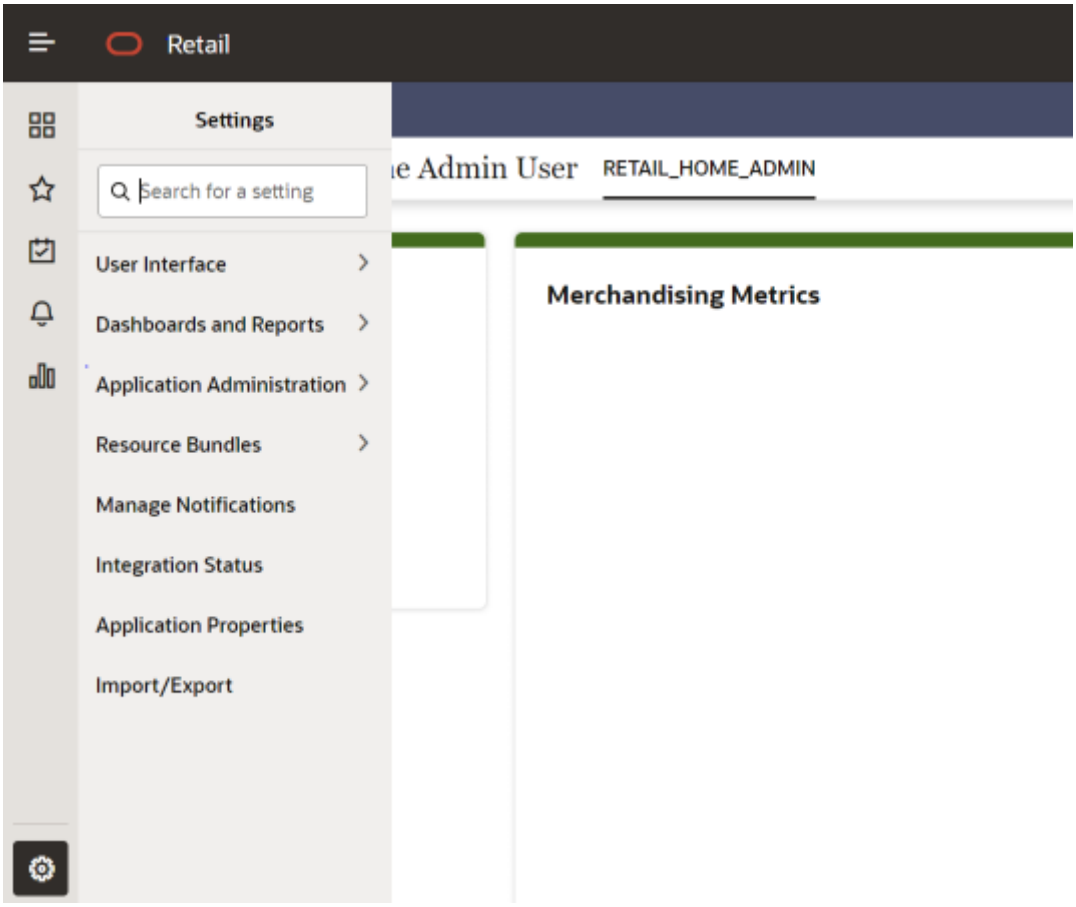
Create OAuth2 Client Application in IDCS

Use Retail Home for creating the client app in IDCS. Once app is created you will get client id and client secret both of them necessary to get access token. Follow the instructions below for generating the access token and making service call using OAuth2 token.

1. Login into retail home as retail home administrator.

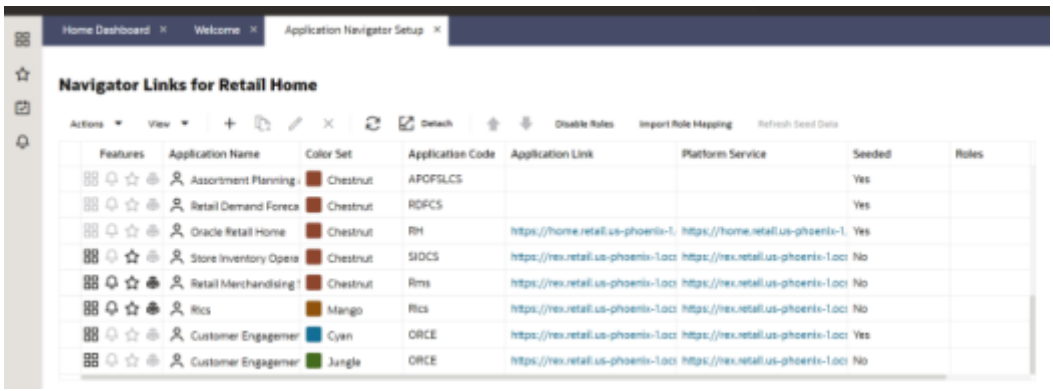


2. In retail home screen click on Settings menu icon on the left and then click on **Application Administration**.



3. On the Application Administra-tion menu click on Application Navigator Setup. Notice all the hosted applications are listed here with their application and plat-form service url.

   Settings -> Application Administration->Application Navigator Settings



4. Look for application with name RICS. If you are not seeing RICS application try refreshing seed. Steps

   a. Select the row with the application code as Rms.

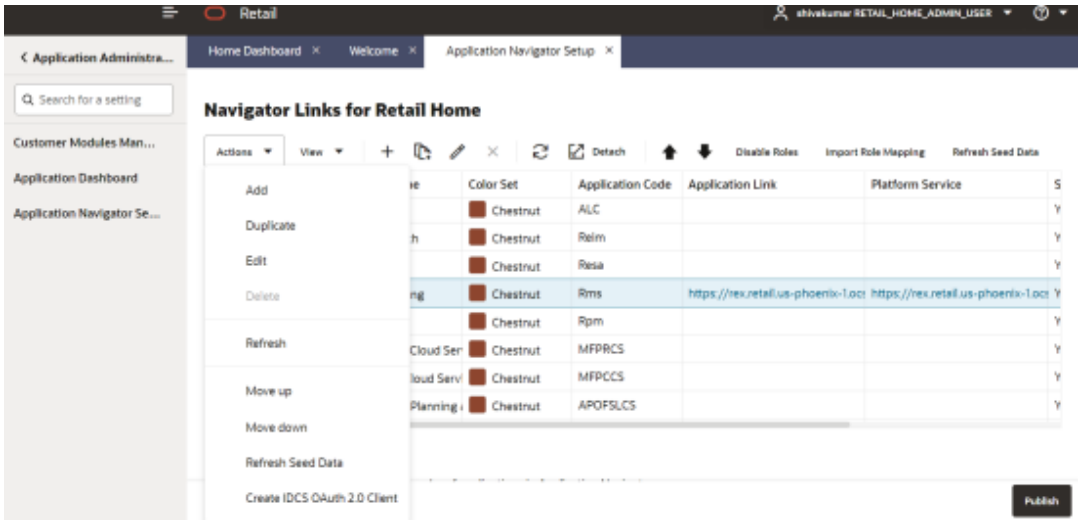
   b. Click the **Refresh Seed Data** button on top right corner of the menu.

   c. Wait for some time and refresh the screen.

   d. RICS should reflect now.

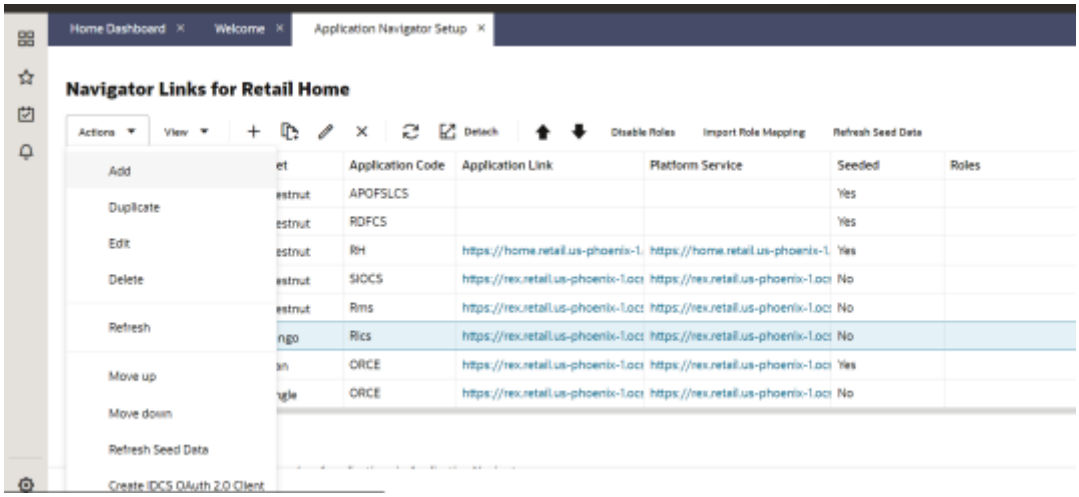
5. If RICS application is not reflecting even after following step 4. Select the row with the application code as Rms and click on the Actions menu on top left. Select Create IDCS OAuth 2.0 Client. A dialog will open for entering oauth2 client details.

> **Note:**
>
> Create IDCS OAuth 2.0 Client option is available only for applications those have platform service URL mentioned. RICS is making use of merch platform service as both the apps are sharing same IDCS tenancy.



6. Skip this step if RICS application is not showing up. One of either Step 5 or Step 6 needs to be followed.

   Select a row with application code as RICS. Click on the Actions menu on top left and select Create IDCS OAuth 2.0 Client. A dialog will open for entering oauth2 client details.



7. This dialog takes the following values:

   App Name is 2-100 characters and will be used as the name in IDCS. Provide unique application name.

   Description is a detailed description of the application.

   Scope: <Custom environment-specific scope>

   The scope pattern that is used in the RICS IDCS app creation template is rgbu:rics:<SERVICETYPE>-<ENVIRONMENT> where SERVICETYPE is RICS and ENVIRONMENT is the environment type (STG, PRD, UAT, DEV1, DEV2, and so on).

   For example:

   ```
   "scope": "rgbu:rics:RICS-PRD""scope": "rgbu:rics:RICS-STG"
   ```

8. When the application is created, another dialog will open to show the client ID and client secret of the new application. These values should be copied down to a safe location, as they will only be shown once. Retail Home cannot retrieve the credentials again after the dialog is closed.

9. Client ID and Client Secret from previous step will be used for generating access token.

   Sample code for generating Access Token:

```
clientId=RICS_TEST_APPID
clientSecret=998e1e1d-f146-45a5-a9a1-99785e3ebf43
idcsUrl=https://idcs-234e8f7334564936aa0ed93f2c39e9ca.identity.pint.oc9qadev.com
scope=rgbu:rics:RICS-STG99
ec=$(echo -n "$clientId:$clientSecret" | base64 -w 0)

AccessToken=$(curl -iv \
-H "Authorization: Basic $ec" \
-H "Content-Type: application/x-www-form-urlencoded;charset=UTF-8" \
--request POST $idcsUrl/oauth2/v1/token \
-d "grant_type=client_credentials&scope=$scope" | grep -o -P '(?
<=access_token":").*(?=","token_type)')

echo $AccessToken
```

10. Now service call can be made by passing along the access token generated in previous step.

    Here is sample curl command with Bearer token and rib-ext publisher ping

```
ribExtServiceUrl=https://rex.retail.us-phoenix-1.ocs.oc-test.com:443/rgbu-rex-eit-
stg99-rics/rib-ext-services-web/resources/publisher/ping
curl -ivkL --noproxy '*' -H "Authorization: Bearer $AccessToken"  -H "Content-Type:
application/xml" -X GET $ri-bExtServiceUrl
```

    Sample response

```
{"message": "ping() was called with input String of: hello"}
```

11. Publishing a message using access token.

    Here is sample curl for publishing a message

```
ribExtServiceUrl=https://rex.retail.us-phoenix-1.ocs.oc-test.com:443/rgbu-rex-eit-
stg99-rics/rib-ext-services-web/resources/publisher/publish
curl -ivkL --noproxy '*' -H "Authorization: Bearer $AccessToken"  -H "Content-Type:
application/xml" -X POST $ribExtServiceUrl --data '<v1:ApplicationMessages
xmlns:v1="http://www.oracle.com/retail/integration/rib/ApplicationMessages/v1">
<v1:ApplicationMessage>
<v1:family>InvAdjust</v1:family>
<v1:type>InvAdjustCre</v1:type>
<v1:payloadXml>&lt;InvAdjustDesc xmlns=&quot;http://www.oracle.com/retail/
integration/base/bo/InvAdjustDesc/v1&quot; xmlns:xsi=&quot;http://www.w3.org/2001/
XMLSchema-instance&quot;
xsi:schemaLocation=&quot;http://www.oracle.com/retail/integration/base/bo/
InvAdjustDesc/v1
http://www.oracle.com/retail/integration/base/bo/InvAdjustDesc/v1/
InvAdjustDesc.xsd&quot
;&gt;&lt;dc_dest_id&gt;DC_ES&lt;/
dc_dest_id&gt;&lt;InvAdjustDtl&gt;&lt;item_id&gt;Aline&lt;/
item_id&gt;&lt;adjustment_reason_code&gt;stri&lt;/
adjustment_reason_code&gt;&lt;unit_qty&gt;22.4&lt;/unit_qty&gt
;&lt;transshipment_nbr&gt;ss&lt;/transshipment_nbr&gt;&lt;from_disposition&gt;ss&lt;/
from_disposition&gt;&lt;to_disposition&gt;sss&lt;/
to_disposition&gt;&lt;from_trouble_code&gt;sss&lt;/from_trouble_code&gt;
&lt;to_trouble_code&gt;ss&lt;/to_trouble_code&gt;&lt;from_wip_code&gt;aaa&lt;/
from_wip_code&gt;&lt;to_wip_code&gt;sss&lt;/
to_wip_code&gt;&lt;transaction_code&gt;4&lt;/
transaction_code&gt;&lt;user_id&gt;TestUser&lt;/user_id&gt;
&lt;create_date&gt;1999-10-23T20:27:56.32&lt;/
create_date&gt;&lt;po_nbr&gt;PratapOrd96&lt;/po_nbr&gt;&lt;doc_type&gt;P&lt;/
doc_type&gt;&lt;aux_reason_code&gt;string&lt;/aux_reason_code&gt;
&lt;weight&gt;12.4&lt;/weight&gt;&lt;weight_uom&gt;smn;&lt;/
weight_uom&gt;&lt;unit_cost&gt;20.4&lt;/
unit_cost&gt;&lt;InvAdjustUin&gt;&lt;uin&gt123&lt;/uin&gt;
&lt;status&gt;4&lt;/status&gt;&lt;/InvAdjustUin&gt;&lt;/InvAdjustDtl&gt;&lt;/
InvAdjustDesc&gt;</v1:payloadXml>
</v1:ApplicationMessage>
</v1:ApplicationMessages>'
```

Sample response

```
{"message": "Publish done"}
```

External Application as a Subscriber (rest-app)

For an external application to consume the message from the RIB's JMS on cloud, it has to host the Injector Service. Injector Service is a ReST webservice that is made available as a pluggable jar.

A pluggable jar is provided which contains all the wrapper classes to help in implementing injector service. rib-injector-services-web-<version>.war is the pluggable jar which can be included into the external application deployable file for example, ext-app.ear/lib. Once pluggable jar is added, endpoint for injector service will be exposed as follows:

```
https://<external-app-host>:<port>/ rib-injector-services-web/resources/injector/inject
```

```
Pluggable jar is provided for reference however customer can choose to write their own
injector service by adhering to REST service contract detailed in next section.
```

> **Note:**
>
> For information on pluggable jar, see the Client Connector For Oracle Retail Integration Cloud Service 24.0.201.0 (Patch) available on My Oracle Support.

How to implement Injector Service (Service Contract) using ReST

Here is the Rest service contract detail:

1. Keep the path as Injector/inject.

   ```
   @Path("/injector")
   ```

2. Use POST for this service. As the input message object itself has identifier (message type-CRE/MOD) they don't need to use the PUT/PATCH. they can use message type to build the implementation logic.

   ```
   @POST
   @Path("/inject")
   @Consumes({MediaType.APPLICATION_XML})
   ```

3. The input would be MediaType.APPLICATION_XML and the structure would be 'ApplicationMessage' object. (file attached for reference).

   ```
   <xs:element name="ApplicationMessage">
     <xs:complexType>
       <xs:sequence>
         <xs:element name="family" type="string25"/>
         <xs:element name="type" type="string30"/>
         <xs:element name="businessObjectId" type="string255" minOccurs="0"/>
         <xs:element ref="ApplicationMessageRoutingInfo" minOccurs="0"
   maxOccurs="unbounded"/>
         <xs:element name="payloadXml" type="xs:string"/>
       </xs:sequence>
     </xs:complexType>
   </xs:element>
   ```

4. Customer can utilize the payload.properties file for validation of message family and type.
5. Return type should be JSON, see below example:

   ```
   String message = "{\"message\": \"Inject successful.\"}";
   return Response.ok(message, MediaType.APPLICATION_JSON).build();
   ```

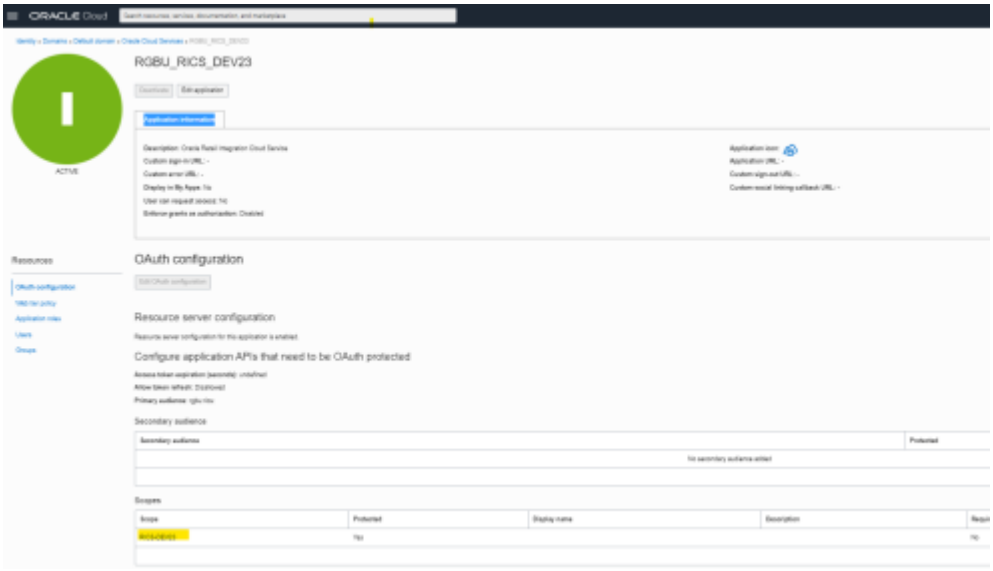
6. For exception response customer needs to follow the structure of exceptionVO.

How to Secure Injector Service with Oauth2

Injector service exposed by external service should be secured with OAuth2. This chapters covers the key points that should be taken into consideration while protecting the resources exposed by external application.

Prerequisites

- IDCS should be same as RICS.
- Use Client Credentials grant type with scope to provide access to resource.
- Following is the screen shot of a sample IDCS app with scope added



> **Note:**
>
> Follow IDCS documentation for detailed instruction on setup.

RIB-EXT Side of Configuration to Point to External Application

Below are the steps to point rib-ext to the correct injector service.

Table 8-2

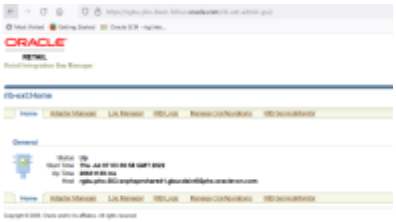
Category	Step	Comment
Access RIB Admin GUI	Access the rib admin GUI at https://<external-load-balancer>/rib-ext-admin-gui Log in with the admin user.	

Table 8-2 (Cont.)

Category	Step	Comment
Verify Configuration and update	Navigate to Manage Configurations -> System options Search for and verify the following: 1. destination.retail.appType : rest-app 2. Update the value for InjectorService URL (injector.service.endpoint.url). URL should point to inject service provided by external application. (e.g.- https://<host:port>/rib-injector-services-web/resources/injector/inject 3. Update the value for Ping Service URL (injector.service.endpoint.ping.url). This URL should point to a ping service provided by an external application. (for example - `https://<host:port>/rib-injector-services-web/resources/injector/ping`). **Note:** This feature allows users to provide their ping URL, as it can be a freeform URL. Previously, it was assumed ping used `host:port/injector/ping`, causing ping to fail when the systems used its own ping URL. Now, the ping feature in rib-ext relies on the ping implemented on the system. Ping is typically used to test the first-time handshake between the service client and the service provider before sending the actual data over to OIC. The fact that data is moving to OIC tells us that the integration is working fine. 4. For 3rd party integration where the injector service	

Table 8-2 (Cont.)

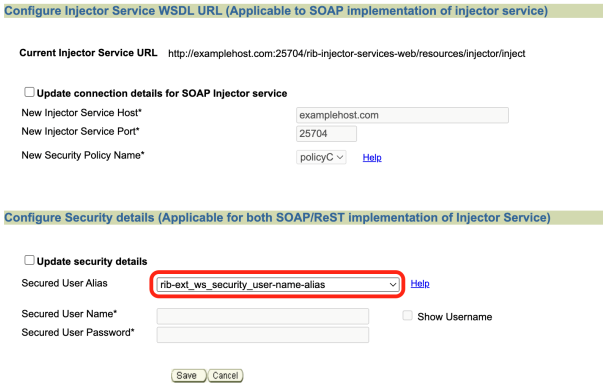
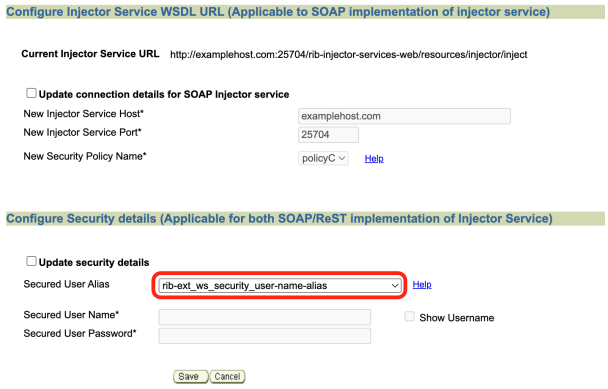
Category	Step	Comment
	is hosted on OIC/on-prem, the below system property in JAVA_OPTIONS needs to be added oauth2.url.path.wo.vrc=< context root of injector service> **Important:** context root of injector service is any word in injector service url which can identify service uniquely. Eg- For the following injector service url https://<external-lb>/ external-injector-services/external/ribinjector/inject. Java_option would be oauth2.url.path.wo.vrc=ribinjector 5. Security Policy (injector.service.security.policyname) : policyA 6. IDCS OAuth Server URL (oauth2.default.authorizationServerUrl): https://<idcs-tenant>/oauth2/v1/token 7. OAuth2 Token Scope: Update with external application provided scope	
Update username and password to empty	Navigate to Manage Configurations - > Injector Service Update details. 1. Choose "rib-(app)_ws_security_user-name-alias" as Secured User Alias. 2. Update the Secured User Name with a blank userName. 3. Update the Secured User Password with a blank password. 4. Click on Save.	

Table 8-2 (Cont.)

Category	Step	Comment
Update ClientID/ Secret	Navigate to Manage Configurations - > Injector Service Update details 1. Choose "rib-(app)_oauth2_application_client_user-name-alias" as Secured User Alias. 2. Update the Secured User Name with clientID. 3. Update the Secured User Password with clientSecret.	
Ping Test	Navigate to Manage Configurations -> RIB Service Monitor 1. Click on ping 2. It should return success	

How to verify provided injector service details are correct	Verify if the provided injector service URL and credentials are correct.	Execute the following curl commands
		`ClientId=56c7eb72f11b43bb98bf2570fa2353eb`
		`ClientSecret=bb18aa22-4bb4-41d1-9ed4-fea276651e28`
		`IDCSUrl=https://idcs-24e4baae56764e91be371e6a2060d66e.identity.c9dev2.oc9qadev.com`
		`AccessToken=$(curl -i -X POST \ --user $ClientId:$ClientSecret \ -H "Content-Type: applica-tion/x-www-form-urlencoded;charset=UTF-8" \ $IDCSUrl/oauth2/v1/token \ -d "grant_type=client_credentials&scope=urn:opc:idm:__myscopes__" | grep -o -P '(?<=access_token":").*(?=","token_type)')`
		`ribExtServiceUrl=https://rgbu-phx-lbext-351.us.oracle.com/rib-injector-services-web/resources/injector/ping`
		`curl -ivkL --noproxy '*' -H "Authorization: Bearer $AccessToken" -H "Content-Type: applica-tion/xml" -X GET $ribExtServiceUrl`

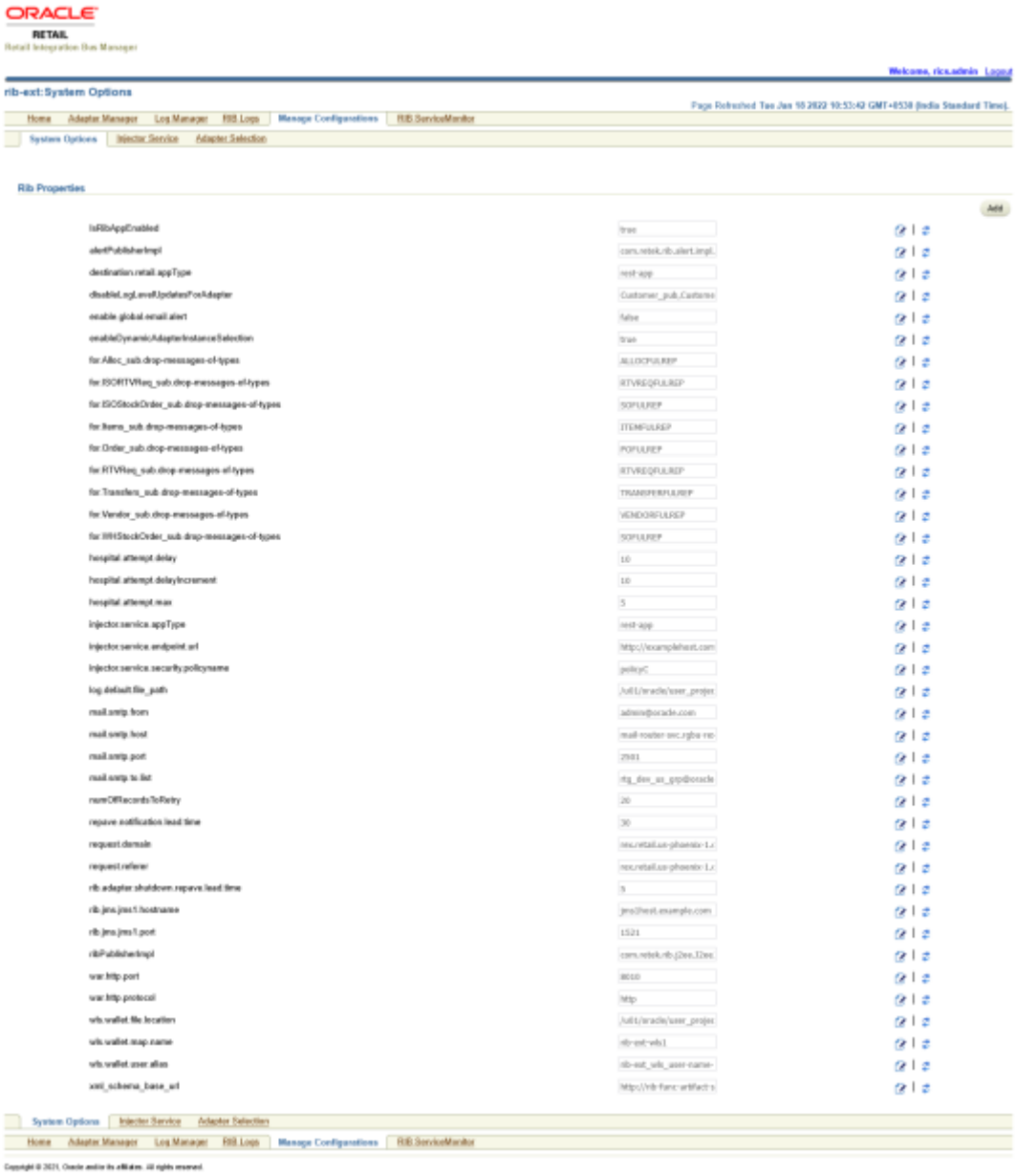
How to switch Injector Service app Type at Runtime

RIB-EXT is a rest-app by default for CFS and expects injector service also to be of ResT type. ONLY for egress/migration customers who already have injector service SOAP implementation in GBUCS they should follow these steps to switch from rest to soap based injector calls and vice-versa.

How to Change rib-ext injector-service-app-type from REST to SOAP

1. Open rib-ext admin gui. Go to Manage Configurations > System Options, observe new prop-erty i.e. injector-service-appType added to allow switching injector service app-type at runtime.

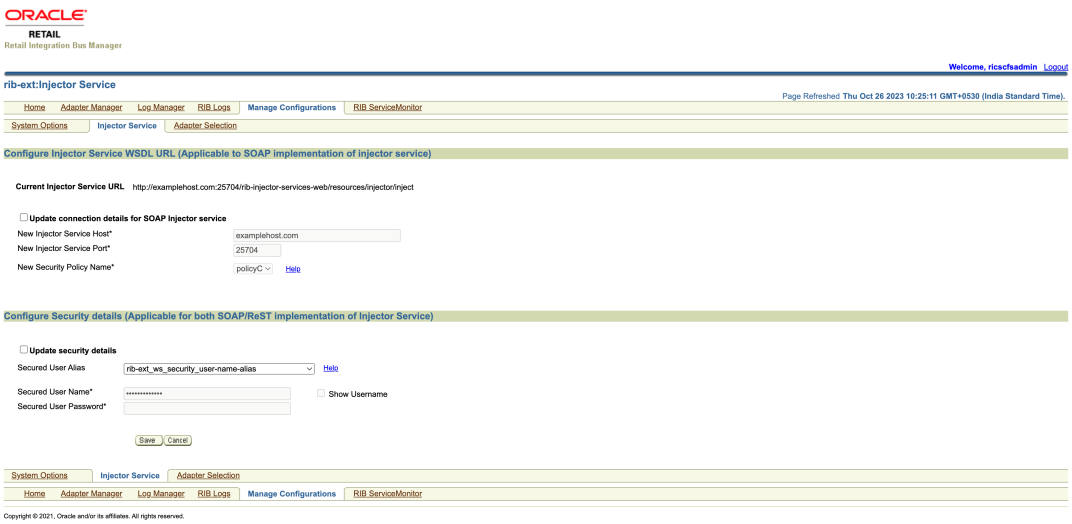
   By default rib-ext is deployed as rest-app so injector-service-appType is defaulted to.



2. Edit injector-service-appType and update this to soap-app. Save the changes.

3. Navigate to Manage Configurations > Injector Service tab. Check for the correctness of injector service URL, ensure it points to correct ext-app injector service.

   Update rib-ext_ws_security_user-name-alias with correct username/password needed to make inject call.



4. Update the value for the Ping Service URL (`injector.service.endpoint.ping.url`). This URL should point to a ping service WSDL provided by an external application.

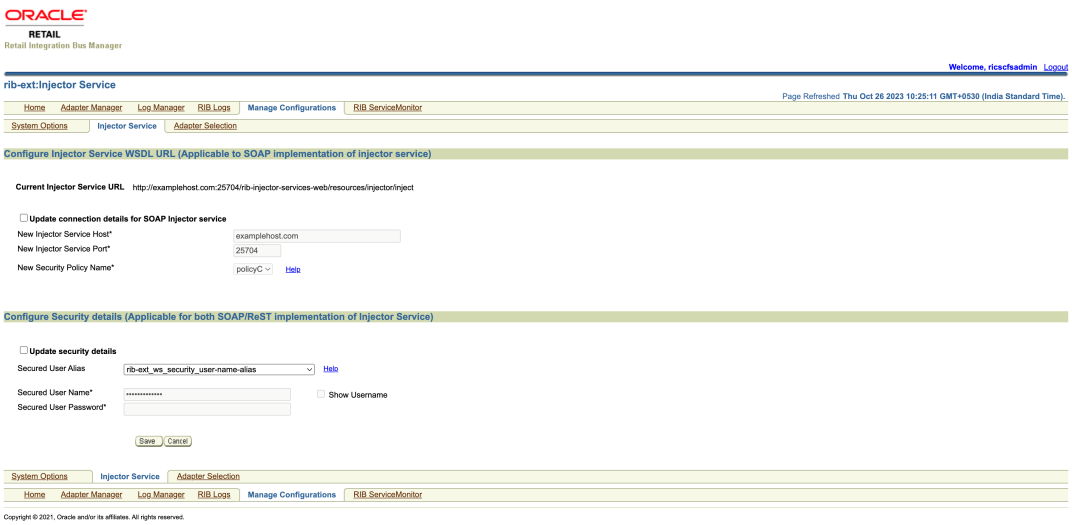
> **Note:**
>
> This feature allows users to provide their ping URL. The ping feature in `rib-ext` relies on the ping implemented on the system. Ping is typically used to test the first-time handshake between the service client and the service provider before sending the actual data to OIC. The fact that data is moving to OIC tells us that the integration is working fine.

5. Setup is ready now. Do a ping test from RIB ServiceMonitor tab.

How to change rib-ext injector-service-app-type from SOAP to ReST

1. Navigate to Manage Configurations > System Options from admin GUI. Look for injector-service-appType, update this property to switch from SOAP to ReST.Save the changes.

2. Navigate to Injector Service tab. Update host/port and security credentials (rib-ext_ws_security_user-name-alias) if needed.



3. Setup is ready now. Do a ping test from RIB Service Monitor tab.

Error Handling

The RIB infrastructure provides a mechanism called RIB error hospital to handle and manage the error messages. When the publishing or subscription of a message fails in the rib-ext for some reason, it lands in error hospital with a reason code. The retry adapters in the rib-ext application are responsible for retrying the messages in error hospital.

Oracle RIB Hospital Administration (RIHA) is a Weblogic application that allows the management of messages in error hospital. Some of the RIHA operations include:

- Viewing error messages
- Editing error messages
- Retrying error messages
- Stopping error messages

For more information, see the Oracle Retail Integration Bus Hospital Administration Guide.

Monitoring Integration

To monitor live statistics of various components involved in RIB integration system like RIB adapter, error hospital, JMS server, RTG provides a live monitoring application called the Retail Integration Console (RIC).

The RIC is the user interface application designed to provide a unified view of the RTG integration products within the business context of the Oracle Retail applications. It provides near real time statistics regarding the message flows, JMS topics, historical trends of each message family, performance comparisons, and static information like application configuration.

For more information, see the Oracle Retail Integration Console User Guide.

A

Appendix - Sample Files

Sample Application.wadl File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ns0:application xmlns:ns0="http://wadl.dev.java.net/2009/02">
  <ns0:doc ns1:generatedBy="Jersey: 2.22.4 2016-11-30 13:33:53" xmlns:ns1="http://
jersey.java.net/"/>
  <ns0:doc ns2:hint="This is simplified WADL with user and core resources only. To get
full WADL with extended resources use the query parameter detail. Link: http://
abc.us.oracle.com:8003/rib-injector-services-web/resources/application.wadl?detail=true"
xmlns:ns2="http://jersey.java.net/"/>
  <ns0:grammars>
    <ns0:include href="application.wadl/xsd0.xsd">
      <ns0:doc title="Generated" xml:lang="en"/>
    </ns0:include>
  </ns0:grammars>
  <ns0:resources base="http://abc.us.oracle.com:8003/rib-injector-services-web/
resources/">
    <ns0:resource path="discover">
      <ns0:method id="discoverAllResources" name="GET">
        <ns0:response>
          <ns0:representation mediaType="application/json"/>
        </ns0:response>
      </ns0:method>
    </ns0:resource>
    <ns0:resource path="/injector">
      <ns0:resource path="/inject">
        <ns0:method id="injectMessage" name="POST">
          <ns0:request>
            <ns0:representation mediaType="application/xml"
element="ns3:ApplicationMessage" xmlns:ns3="http://www.oracle.com/retail/integration/rib/
ApplicationMessages/v1"/>
          </ns0:request>
          <ns0:response>
            <ns0:representation mediaType="*/*"/>
          </ns0:response>
        </ns0:method>
      </ns0:resource>
      <ns0:resource path="/ping">
        <ns0:method id="ping" name="GET">
          <ns0:request>
            <ns0:param name="pingMessage" default="hello" type="xsd:string"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" style="query"/>
          </ns0:request>
          <ns0:response>
            <ns0:representation mediaType="application/json"/>
          </ns0:response>
        </ns0:method>
      </ns0:resource>
    </ns0:resource>
  </ns0:resources>
</ns0:application>
```

Sample Resource Class

```
package com.oracle.retail.rib.integration.services.applicationmessageinjector;

import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import com.oracle.retail.integration.rib.applicationmessages.v1.*;
import com.retek.rib.binding.exception.InjectorException;
import com.retek.rib.binding.injector.Injector;
import com.retek.rib.binding.injector.InjectorFactory;
import com.retek.rib.domain.payload.PayloadFactory;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import com.oracle.retail.integration.payload.Payload;

@Stateless
@Path("/injector")

public class ApplicationMessageInjectorResource {

    private static Log LOG =
            LogFactory.getLog(ApplicationMessageInjectorResource.class);

    @GET
    @Path("/ping")
    @Produces({MediaType.APPLICATION_JSON})
    public Response ping(@DefaultValue("hello") @QueryParam("pingMessage") String
pingMessage){
        String message = "{\"message\": \"Got " + pingMessage + " from server.\"}";
        return Response.ok(message, MediaType.APPLICATION_JSON).build();
    }

    @POST
    @Path("/inject")
    @Consumes({MediaType.APPLICATION_XML})
    public Response injectMessage(ApplicationMessage applicationMessage) throws
InjectorException{

        verifyNotNull(applicationMessage, "applicationMessage");


        invokeInjectForMessageType(applicationMessage.getFamily(),
applicationMessage.getType(), applicationMessage.getBusinessObjectId(),
applicationMessage.getPayloadXml());

        String message = "{\"message\": \"Inject successful.\"}";
        return Response.ok(message, MediaType.APPLICATION_JSON).build();
    }


    private void invokeInjectForMessageType(String family, String messageType, String
```

```
businessObjectId, String retailPayload)throws InjectorException{

        try {

            verifyNotNull(family, "family");
            verifyNotNull(messageType, "messageType");
            verifyNotNull(retailPayload, "retailPayload");

            Payload payload = PayloadFactory.unmarshalPayload(family, messageType,
retailPayload);

            Injector injector = InjectorFactory.getInstance().getInjector(
      ??              family, messageType);
            if (injector == null) {
                final String eMsg = "Unknown message"
                    + " family/type: " + family + "/" + messageType;
                LOG.error(eMsg);
                throw new InjectorException(eMsg);

            }
            if(LOG.isDebugEnabled()){
                LOG.debug("Received inject call for family("+family+")
type("+messageType+") businessObjectId("+businessObjectId+") with payload:\n" +
payload.toString());
            }

            injector.inject(messageType, businessObjectId, payload);
            LOG.debug("Inject call for family("+family+") type("+messageType+")
businessObjectId("+businessObjectId+") return.");

  ??     } catch (InjectorException e) {
            final String eMsg = "Exception calling inject.";
            LOG.error(eMsg, e);
            throw e;
        }catch (Exception re) {
            final String eMsg = "Exception calling inject.";
            LOG.error(eMsg, re);
            throw new RuntimeException(eMsg, re);
        }

    }

    private void verifyNotNull(Object field, String fieldName){
      if(field == null){
        final String eMsg = fieldName + " cannot be null.";
        LOG.error(eMsg);
        throw new IllegalArgumentException(eMsg);
      }
    }

}
```

ApplicationMessages.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://www.oracle.com/retail/integration/rib/ApplicationMessages/v1"
           xmlns:rib="http://www.oracle.com/retail/integration/rib/
ApplicationMessages/v1"
           xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
```

```
            jaxb:extensionBindingPrefixes="xjc"
            jaxb:version="2.0"
            targetNamespace="http://www.oracle.com/retail/integration/rib/
ApplicationMessages/v1"
            elementFormDefault="qualified" attributeFormDefault="unqualified">

    <xs:annotation>
        <xs:appinfo>

            <jaxb:globalBindings
                fixedAttributeAsConstantProperty="false"
                choiceContentProperty="true"
                enableFailFastCheck="true"
                generateIsSetMethod="true"
                enableValidation="true">
                <!--xjc:javaType name="java.util.Calendar"
                              xmlType="xs:dateTime"

adapter="com.oracle.retail.integration.rib.rib_integration_runtime_info.datatypeadapter.C
alendarAdapter"/ -->
                <jaxb:serializable uid="1"/>
            </jaxb:globalBindings>

            <!--jaxb:schemaBindings>
                <jaxb:package
name="com.oracle.retail.integration.rib.ribintegrationruntimeinfo" />
            </jaxb:schemaBindings-->
        </xs:appinfo>
    </xs:annotation>

    <xs:element name="ApplicationMessages">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="ApplicationMessage" maxOccurs="unbounded" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="ApplicationMessage">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="family" type="string25"/>
                <xs:element name="type" type="string30"/>
                <xs:element name="businessObjectId" type="string255"
 minOccurs="0"/>
                <xs:element ref="ApplicationMessageRoutingInfo" minOccurs="0"
maxOccurs="unbounded"/>
                <xs:element name="payloadXml" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="ApplicationMessageRoutingInfo">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="string25"/>
                <xs:element name="value" type="string25"/>
                <xs:element ref="ApplicationMessageRoutingInfoDetail" minOccurs="0"
maxOccurs="2"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
```

```
  <xs:element name="ApplicationMessageRoutingInfoDetail">
      <xs:complexType>
          <xs:sequence>
              <xs:element name="name" type="string25"/>
              <xs:element name="value" type="string300"/>
          </xs:sequence>
      </xs:complexType>
  </xs:element>

 <xs:simpleType name="string255">
   <xs:restriction base="xs:string">
     <xs:maxLength value="255" />
   </xs:restriction>
 </xs:simpleType>

 <xs:simpleType name="string25">

  <xs:restriction base="xs:string">
     <xs:maxLength value="25" />
   </xs:restriction>
 </xs:simpleType>

 <xs:simpleType name="string30">
   <xs:restriction base="xs:string">
     <xs:maxLength value="30" />
   </xs:restriction>
 </xs:simpleType>

   <xs:simpleType name="string300">
   <xs:restriction base="xs:string">
     <xs:maxLength value="300" />
   </xs:restriction>
 </xs:simpleType>


</xs:schema>

payload.properties
```

payload.properties

```
ASNIN.ASNINCRE=com.oracle.retail.integration.base.bo.asnindesc.v1.ASNInDesc
ASNIN.ASNINDEL=com.oracle.retail.integration.base.bo.asninref.v1.ASNInRef
ASNIN.ASNINMOD=com.oracle.retail.integration.base.bo.asnindesc.v1.ASNInDesc

WH.WHCRE=com.oracle.retail.integration.base.bo.whdesc.v1.WHDesc
WH.WHDEL=com.oracle.retail.integration.base.bo.whref.v1.WHRef
WH.WHMOD=com.oracle.retail.integration.base.bo.whdesc.v1.WHDesc
```

Sample Request/Response for ReST Injector Service

Table A-1 Sample Request/Response for ReST Injector Service

End Point	Method	Media Type	User/ Password	Request.xml	Response	Comments
http://localhost:7001/rib-injector-services-web/resources/injector/inject	POST	application/xml Request are xml only and response are json only.	A valid user that is part of IntegrationGroup.	<v1:ApplicationMessage xmlns:v1="http://www.oracle.com/retail/integration/rib/ApplicationMessages/v1"> <v1:family>XOrder</v1:family> <v1:type>XOrderCre</v1:type> <v1:businessObjectId>592824510</v1:businessObjectId> <v1:payloadXml><<XOrderDesc xmlns="http://www.oracle.com/retail/integration/base/bo/XOrderDesc/v1" xmlns:ns0="http://www.oracle.com/retail/integration/base/bo/CustFlexAttriVo/v1"> <order_no>592824510</order_no> <supplier>99</supplier> <currency_code>USD</currency_code> <terms>13</terms> <not_before_date>2022-02-09T00:00:00Z</not_before_date> <not_after_date>2022-02-19T00:00:00Z</not_after_date> <otb_eow_date&	HTTP/1.1 200 OK Date: Thu, 10 May 2018 16:33:11 GMT Content-Length: 33 Content-Type: application/json X-ORACLE-DMS-ECID: 4a8e5d3f-1aae-43d7-ba84-c6b9c60563c7-00000039 X-ORACLE-DMS-RID: 0 Set-Cookie: JSESSIONID=hsFK5jW4B1QtipC9zhng--or1WL7ywxCuxsJeVwdgPpnv6oNUnde!233126712; path=/; HttpOnly {"message": "Inject successful."}	Success

Table A-1 (Cont.) Sample Request/Response for ReST Injector Service

End Point	Method	Media Type	User/ Password	Request.xml	Response	Comments
				gt;2022-02-19T00: 00:00Z</ otb_eow_date> <status>A&l t;/status> <exchange_rate >1</ exchange_rate> <include_on_or d_ind>Y</ include_on_ord_in d> <written_date& gt;2022-02-09T00: 00:00Z</ written_date> <XOrderDtl> <item>17425 0093</item> <location>2 1</ location> <unit_cost> 10</ unit_cost> <origin_countr y_id>US</ origin_country_id > <supp_pack_siz e>1</ supp_pack_size&gt ; <qty_ordered&g t;2</ qty_ordered> <location_type >W</ location_type> <reinstate_ind >N</ reinstate_ind> <delivery_date >2022-02-09T00 :00:00Z</ delivery_date> </ XOrderDtl> <orig_ind>2 </orig_ind> <edi_po_ind&gt ;N</ edi_po_ind> <pre_mark_ind&		

Table A-1 (Cont.) Sample Request/Response for ReST Injector Service

End Point	Method	Media Type	User/ Password	Request.xml	Response	Comments
				gt;N</ pre_mark_ind> </ XOrderDesc>></ v1:payloadXml> </ v1:ApplicationMes sage>		

Table A-1 (Cont.) Sample Request/Response for ReST Injector Service

End Point	Method	Media Type	User/ Password	Request.xml	Response	Comments
			If user in not added in IntegrationGroup	<v1:ApplicationMessage xmlns:v1="http://www.oracle.com/retail/integration/rib/ApplicationMessages/v1"> <v1:family>WH</v1:family> <v1:type>WHCR</v1:type> <!--Optional:--> <v1:businessObjectId>?</v1:businessObjectId> <!--Zero or more repetitions:--> <v1:ApplicationMessageRoutingInfo> <v1:name>?</v1:name> <v1:value>?</v1:value> <!--Zero or more repetitions:--> <v1:ApplicationMessageRoutingInfoDetail> <v1:name>?</v1:name> <v1:value>?</v1:value> </v1:ApplicationMessageRoutingInfoDetail> </v1:ApplicationMessageRoutingInfo> <v1:payloadXml><WHDesc xmlns="http://www.oracle.com/retail/integration/base/bo/WHDesc/v1"><wh>10</wh><wh_name>g</wh_name></WHDesc></	HTTP/1.1 403 Forbidden Date: Thu, 05 Aug 2021 10:25:26 GMT Content-Length: 1166 Content-Type: text/html; char-set=UTF-8 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Draft//EN"> <HTML> <HEAD> <TITLE>Error 403--Forbidden</TITLE> </HEAD> <BODY bgcol-or="white"> <FONT FACE=Helvetica><BR CLEAR=all> <TABLE bor-der=0 cellspac-ing=5><TR><TD><BR CLEAR=all> <FONT FACE="Helvetica" COL-OR="black" SIZE="3"><H2>Erro r 403--Forbidden</H2> </FONT></TD></TR> </TABLE> <TABLE bor-der=0 width=100% cellpad-ding=10><TR><TD VALIGN=top WIDTH=100% BGCOL-OR=white><FONT FACE="Courier New"><FONT FACE="Helvetica" SIZE="3"><H3>From RFC 2068 Hypertext Transfer Protocol -- HTTP/1.1:</H3>	Failure

Table A-1 (Cont.) Sample Request/Response for ReST Injector Service

End Point	Method	Media Type	User/ Password	Request.xml	Response	Comments
				v1:payloadXml> </ v1:ApplicationMessage>	</FONT><FONT FACE="Helvetica" SIZE="3"><H4>10.4 .4 403 Forbidden</H4> </FONT> <P><FONT FACE="Courier New">The server understood the request, but is refusing to fulfill it. Authorization will not help and the request SHOULD NOT be repeated. If the request method was not HEAD and the server wishes to make public why the request has not been ful-filled, it SHOULD de-scribe the reason for the refusal in the entity. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is ap-plica-ble.</ FONT></P> </FONT></TD></TR> </TABLE> </BODY> </HTML>	

Table A-1 (Cont.) Sample Request/Response for ReST Injector Service

End Point	Method	Media Type	User/ Password	Request.xml	Response	Comments
			Wrong User/ pass	<v1:ApplicationMessage xmlns:v1="http://www.oracle.com/retail/integration/rib/ApplicationMessages/v1"> <v1:family>WH</v1:family> <v1:type>WHCR</v1:type> <!--Optional:--> <v1:businessObjectId>?</v1:businessObjectId> <!--Zero or more repetitions:--> <v1:ApplicationMessageRoutingInfo> <v1:name>?</v1:name> <v1:value>?</v1:value> <!--Zero or more repetitions:--> <v1:ApplicationMessageRoutingInfoDetail> <v1:name>?</v1:name> <v1:value>?</v1:value> </v1:ApplicationMessageRoutingInfoDetail> </v1:ApplicationMessageRoutingInfo> <v1:payloadXml><WHDesc xmlns="http://www.oracle.com/retail/integration/base/bo/WHDesc/v1"><wh>10</wh><wh_name>g</wh_name></WHDesc></	HTTP/1.1 401 WWW-Authenticate: Basic realm="Authentication required" Content-Type: text/html;charset=utf-8 Content-Language: en Content-Length: 669 Date: Thu, 05 Aug 2021 05:08:40 GMT Keep-Alive: timeout=20 Connection: keep-alive <!doctype html><html lang="en"><head><title>HTTP Status 401 â€" Unauthorized</title><style type="text/css">body {font-family:Tahoma,Arial,sans-serif;} h1, h2, h3, b {color:white;background-color:#525D76;} h1 {font-size:22px;} h2 {font-size:16px;} h3 {font-size:14px;} p {font-size:12px;} a {color:black;} .line {height:1px;background-color:#525D76;border:none;}</style></head><body><h1>HTTP Status 401 â€"	Failure

Table A-1 (Cont.) Sample Request/Response for ReST Injector Service

End Point	Method	Media Type	User/ Password	Request.xml	Response	Comments
				v1:payloadXml> </ v1:ApplicationMessage>	Unauthorized</ h1><hr class="line" /> Type Status Report</ p> Description The request has not been applied because it lacks valid authentication credentials for the target resource. <hr class="line" /><h3>Apache Tomcat/8.5.64</ h3></body></html>	