28
Developing Applications with Sagas

This chapter covers Saga APIs that allow applications that are built using microservices principles to incorporate efficient transactions across microservices. Saga APIs are implemented while closely following the Long Running Action (LRA) specification from Eclipse MicroProfile.

Topics:

28.1 Implementing Sagas with Oracle Database

You can use Oracle Database as a platform to build your Saga-based, on-cloud or on-premises applications. When you implement Sagas with Oracle Database for handling distributed transactions, you can leverage the robust underlying infrastructure built into Oracle Database. The Saga infrastructure maintains the global application state for microservices, facilitating development and throughput.

The following are some key benefits of using Oracle Database for Saga implementation:

- The Saga implementation is integrated in the database to make Sagas simpler to code, deploy, and maintain.
- Simplified Saga annotations following the Eclipse LRA standard make it easy to enable transactions across microservices using Oracle Database.
- Data consistency is achievable across microservices.
- Auto-compensating logic based on lock-free, reservable columns is provided in the database to handle rollbacks on Saga rollback.
- Advanced Queuing (AQ) and Oracle Transactional Event Queues (TxEventQ) messaging platforms are built into Oracle Database, enabling microservices to produce and consume messages and events as part of database transactions.
- Support is provided for different languages, including PL/SQL and Java applications.
- The Saga framework recommends a JSON type for representing the application payload.
- There is improved availability and scalability for applications that require transactions across microservices. Such transactions are common with monolith applications that are

converting to microservices, where transactions may span multiple bounded contexts. Note that a bounded context is an explicit boundary within which a business domain model exists.

28.2 Oracle Saga Framework Overview

The Oracle Saga framework provides the foundation to implement and administer Sagas for microservices applications built using Oracle Database. The framework provides administrative and client interfaces. Administrative interfaces enable Saga applications to configure and manage Saga participants and message brokers. Client interfaces enable applications to initiate, participate, and finalize Sagas. Saga participants are automatically configured with message channels and message propagation. The Saga framework is compensation-aware and provides for automatic rollback of the affected data when a Saga is rolled back. The Saga framework includes PL/SQL packages, dictionary tables, and Advanced Queuing (AQ) integration that facilitate Sagas in the database.

The Oracle Saga framework leverages two important features of Oracle Database. There is a transaction layer with reservable column support that enables recording of appropriate meta-information when reservable columns are updated. The reservable column support invokes compensating transactions automatically to revert the state of the reservable column updates when a Saga is rolled back. Reservable columns are a part of the lock-free reservation feature that Oracle provides to improve transactional concurrency.

Another important layer integrated into the database is the event queues built on the Oracle Advanced Queuing (AQ) technology. AQ provides the asynchronous messaging platform that connects various microservice participants. The Saga framework deploys a hub-and-spoke topology to connect Saga initiators, coordinators, and participants. The Saga message broker acts as the hub in this topology.

See Also:

- Using Lock-Free Reservation
- *Oracle Database Advanced Queuing User's Guide*

28.3 Saga Framework Features

The Saga framework provides the following features for implementing Sagas on Oracle Database:

- PL/SQL administrative interfaces to add and manage active Sagas and Saga participants
- PL/SQL and Java interfaces for applications to interact with the Saga framework, and participate and finalize (commit or rollback) database Sagas
- Asynchronous message communication across multiple participants that contribute to a Saga, facilitated by microservice-specific AQ message propagation infrastructure from Oracle
- Reservable column-based Saga finalization (compensation and completion) support to enable recording of reservable column updates, and auto-compensation of transactions to roll back changes for canceled Sagas
- User-defined Saga finalization (compensation and completion) support to enable users to explicitly define and automatically execute Saga finalization

- Strong messaging semantics to ensure that interservice messages are delivered exactly once and never lost or duplicated
- Saga ID support to bind individual transactions to the Saga ID
- A globally unique name for each participant in a Saga
- Automatic configuration of message channels and message propagation for Saga participants that are provisioned into the system
- Various dictionary tables to maintain the status of Sagas
- System-defined views to report on Sagas and Saga participants in the system
- Eclipse Microprofile LRA specification emulation

28.4 Saga Framework Concepts

Saga *Participant* and *Initiator*

A Saga participant represents a spoke (participant microservice) in the Saga topology that either initiates and orchestrates a Saga or enrolls to participate in one. A participant is associated with a message broker (that is local or remote) and optionally with a local Saga coordinator. Saga participants that initiate Sagas are described in this document as "initiators" and non-initiator participants are described as "participants". Saga initiators must be associated with a Saga coordinator. In the initial version, the Saga framework only supports a local Saga coordinator (from the same PDB) for an initiator. Meanwhile, a Saga participant need not be associated with a coordinator and can be remote to the initiator.

Any participant can initiate a Saga. A Saga initiator is a microservice that initiates a Saga and enrolls other microservice participants by sending asynchronous messages.

In the travel agency example that follows, the participant microservices are the flight service, hotel service, and car service. A participant microservice executes one or more participant transactions on behalf of a Saga. Compensation action for a participant transaction is maintained in the local PDB. Participants are responsible for maintaining the local state of their Sagas.

Transaction Coordinator

A transaction coordinator acts as a transaction manager in the Saga topology. The Saga coordinator maintains the Saga state on behalf of the Saga initiator.

A Saga topology can have several Saga coordinators. In the initial release of the Saga framework, a Saga participant can only associate with coordinators who are local (same PDB and schema) to the participants. A coordinator can, however, associate with a local or remote message broker.

Message Broker

A message broker acts as an intermediary and represents the hub in the Saga topology. A broker provides a message delivery service for the message propagation between two or more Saga participants and their coordinator. Each Saga participant or coordinator is associated with a single broker, who can be local or remote. A broker does not maintain any Saga state.

Administrator and Client Interfaces

The Saga framework provides two PL/SQL packages: `DBMS_SAGA_ADM` and `DBMS_SAGA`.

The `DBMS_SAGA_ADM` package provides the administrative interface for the Saga framework. Using the administrative interface, you can manage Saga entities and ongoing Sagas.

The DBMS_SAGA package provides the developer APIs to initiate and complete Sagas, when building microservices applications.

Java developers, who want to create microservices using Sagas, should use Saga annotations, which is defined separately in this document.

Saga Annotations

Java applications initiating and participating in Sagas should use the Saga annotations for flexibility and simplicity of development. For more details, see Developing Java Applications Using Saga Annotations section of this document.

Advanced Queuing

The Saga Infrastructure leverages existing Oracle Advanced Queuing (AQ) technology, including message propagation between queues and event notification. AQ is the transaction communication mechanism for Saga participants and coordinators. AQ provides necessary plumbing and message channels to facilitate asynchronous message communication across multiple participants contributing to a Saga. AQ provides exactly-once message propagation without distributed transactions.

Reservable columns

A reservable column is a column type that has auto-compensating capabilities. Reservable columns enable individual databases to maintain reservation journals that record compensating actions for participant transactions. Compensating actions are automatically executed on the reservable column values in the event of a Saga rollback to revert local transactional changes.

Saga local transactions that the participant microservices execute can modify one or more reservable columns of database tables. Sagas leverage reservable columns to record compensating actions when transactional changes are made and automatically invoke the compensating actions when transactions fail. Compensating actions for reservable columns are recorded in reservation journals. Compensating transactions free up any resources that were locked earlier.

Dictionary Views

The Saga framework provides system-defined views to report on Sagas and Saga participants in the system. System defined views ALL_MICROSERVICES, CDB_MICROSERVICES, DBA_MICROSERVICES, and USER_MICROSERVICES provide the ability to monitor Sagas and Saga participants in the system.

The following views show all participants in the system:

- CDB_SAGA_PARTICIPANTS
- DBA_SAGA_PARTICIPANTS
- USER_SAGA_PARTICIPANTS

The following system-defined views show the dynamic state of ongoing Sagas:

- CDB_SAGAS
- DBA_SAGAS
- USER_SAGAS

The following system-defined views show the state of completed Sagas:

- CDB_HIST_SAGAS

- DBA_HIST_SAGAS
- USER_HIST_SAGAS

Information for completed Sagas is retained for a period of 30 days. You can configure this duration using the saga_hist_retention database parameter.

The following system-defined views show incomplete Sagas:

- CDB_INCOMPLETE_SAGAS
- DBA_INCOMPLETE_SAGAS
- USER_INCOMPLETE_SAGAS

The following views provide details about reservable columns for incomplete Sagas:

- CDB_SAGA_PENDING
- DBA_SAGA_PENDING
- USER_SAGA_PENDING

The following views provide details about each Saga:

- CDB_SAGA_DETAILS
- DBA_SAGA_DETAILS
- USER_SAGA_DETAILS

The following views show the pending finalization actions for Sagas:

- CDB_SAGA_FINALIZATION
- DBA_SAGA_FINALIZATION
- USER_SAGA_FINALIZATION

> **See Also:**
>
> Oracle Database Reference guide

Finalization Methods

Sagas are finalized when all participants commit (complete) or roll back (compensate) their respective participant transactions. Saga commit() or rollback() operations result in the completion of a Saga. The Saga framework enables you to also implement application-specific finalization in addition to the implicit reservable column-based finalization.

Eclipse Microprofile LRA Specification

The Saga framework emulates the Eclipse Microprofile LRA specification and provides equivalent functionality with certain limitations. The initial version of the Saga framework has the following limitations:

- Nested Sagas are not supported.
- Saga initiators and coordinators are co-located.

28.5 Initializing the Saga Framework

Initializing Parameters

In the database initialization parameter file, `init.ora`, the `max_saga_duration` parameter identifies the time in seconds beyond which a Saga can be considered incomplete. The default value for this parameter is 86400 seconds. Any Saga that exceeds this configurable duration is considered incomplete and is liable to termination using the `dbms_saga.rollback_saga()` API. The Saga initiator automatically rolls back Sagas that exceed their duration.

In the `init.ora` file, you can use the `_use_saga_qtyp` parameter to set the message queuing type to be used in the Saga infrastructure. By default, the `_use_saga_qtyp` parameter is set to 0, which means it uses classic AQ queues, but if you want to switch to transactional event queues (TEQ), the `_use_saga_qtyp` parameter must be set to 1. Any change in the `_use_saga_qtyp` parameter must be done before creating the Saga entities: broker, coordinator, or participant. Changing the `_use_saga_qtyp` parameter after creating the Saga entities does not change the type of the queues used in the Saga.

> **See Also:**
>
> *Oracle Database Transactional Event Queues and Advanced Queuing User's Guide*

Roles for Access Control

The Saga framework enables database administrators to provide the necessary privileges to the database users, who can administer and participate in Sagas. Users can have the following roles and privileges:

User Role	Access
`SAGA_ADM_ROLE`	Provides the ability to invoke APIs from the `DBMS_SAGA_ADM` package. This role is required for Saga administrators for the initial setup and provides full access to the `DBMS_SAGA_ADM` API. In the Saga framework setup example, `SAGA_ADM_ROLE` privilege is granted to the mailbox user MB.
`SAGA_PARTICIPANT_ROLE`	This role is required for Saga participant services. Saga primitives can only be invoked by a user that has the `SAGA_PARTICIPANT` role granted to it.
`SAGA_CONNECT_ROLE`	This role is provided to the remote dblink user.

`SYS` owns all the dictionary tables and user-accessible views of the data dictionary.

28.6 Setting Up a Saga Topology

The Saga framework provides administration APIs that the DBAs (with `SAGA_ADM_ROLE` privilege) can use to define and manage the Saga participants, coordinators, and brokers. The Saga PL/SQL package called `SYS.DBMS_SAGA_ADM` implements the Saga administrative interface. A participant must invoke the procedures in the `SYS.DBMS_SAGA_ADM` package from inside its schema and PDB.

The administrative interface provides the following APIs to provision Saga participants and brokers.

- `add_participant()` to add participants and their coordinator
- `add_broker()` to explicitly create a broker
- `add_coordinator()` to explicitly create a coordinator
- `drop_participant()` to drop a participant
- `drop_coordinator()` to drop a coordinator
- `drop_broker()` to drop a broker

Saga participants provisioned to the system are automatically configured with message channels and message propagation. Adding a participant creates a corresponding entry in the `SYS.SAGA_PARTICIPANT$` dictionary table, and creates incoming and outgoing Java topics for the participant. The inbound and outbound Java topics are provisioned with system-generated names that are derived from the participant's name. The following sections have further details about the administrative interface processing.

> **See Also:**
>
> DBMS_SAGA_ADM for a complete description of `SYS.DBMS_SAGA_ADM` package APIs

28.6.1 Adding a Message Broker

In the Saga framework, a message broker acts as a message delivery service that receives messages from the Saga participants and propagates them to the Saga recipients.

The Saga framework uses the `dbms_saga_adm.add_broker()` API to add brokers explicitly to the framework. Creating a message broker creates a single Java topic that acts as the mailbox for the Saga participants. The system-generated name of the Java topic is of the form `SAGA$_<broker_name>_INOUT`, where the message broker name (`broker_name`) is the identifier that is provided as an input to the `add_broker()` call.

The `SAGA_ADM_ROLE` (administrator) role is required to create a Saga message broker.

28.6.2 Adding a Coordinator

A Saga coordinator acts as a transaction manager for Sagas. A Saga coordinator maintains the state of the Saga across various participants.

The Saga framework uses the `dbms_saga_adm.add_coordinator()` API to add coordinators to the framework.

> **Note:**
>
> The `add_coordinator()` API must be called prior to invoking the `add_participant()` API, because `add_participant()` needs a coordinator name as an argument. Example: Saga Framework Setup explains the preparatory and provisioning steps.

The `add_coordinator()` interface executes the following:

- Creates system-defined AQ queues for the coordinator's inbound and outbound message channels.
- Establishes a bidirectional message propagation channel between the coordinator and the message broker.

28.6.3 Adding a Participant

A participant is a named entity that represents an application or a microservice that desires to participate in database Sagas. Adding a participant sets up a bidirectional message propagation channel between the participant and the message broker.

The Saga framework uses the `dbms_saga_adm.add_participant()` API to add participants to the framework.

> **Note:**
>
> Prior to invoking `add_participant()`, complete the pre-requisite steps in the participant and broker databases (PDBs). Example: Saga Framework Setup explains the preparatory and provisioning steps.

The `add_participant()` interface executes the following:

- Creates system-defined Java topics for the inbound and outbound message channels of the participants.
- Establishes a bidirectional message propagation channel between the participant and the message broker.

28.6.4 Managing Participants and Message Brokers

You can drop participants, coordinators, and message brokers using the `drop_participant()`, `drop_coordinator()`, and `drop_broker()` APIs. Dropping a participant, coordinator, or message broker also removes the associated JMS topic.

> **Note:**
>
> - You can drop a participant only if there are no ongoing Sagas and no pending messages in the participant's incoming queue.
> - You can drop a coordinator only if there are no participants associated with the coordinator.
> - You can drop a message broker only if there are no registered participants and no pending messages.

> **See Also:**
>
> DBMS_SAGA_ADM for details of administrative APIs in the `SYS.DBMS_SAGA_ADM` package

28.6.5 Message Propagation

Message propagation transfers messages between Saga entities, namely initiators, participants, and brokers. Adding a participant to the Saga framework sets up message propagation. A message propagation job connects a participant's outbound topic to a broker's INOUT topic and connects a broker's INOUT topic to a participant's inbound topic.

- To propagate a participant's outbound topic to a broker's INOUT topic, the message propagation job uses the dblink from the `dblink_to_broker` parameter of the `add_participant()` interface.
- To propagate a broker's INOUT topic to a participant's inbound topic, the message propagation job uses the dblink from the `dblink_to_participant` of the `add_participant()` interface.

  Propagation of messages to a particular participant happens only for messages destined for that participant.

28.6.6 About Dictionary Tables

A globally unique identifier (GUID) is used to identify every Saga transaction. The `sys.saga_finalization$` dictionary table records individual steps that are required to complete the compensating actions for a Saga.

Several dictionary tables track the state (meta-data) associated with the Saga transactions at the participant database. These tables are:

- `sys.saga_message_broker$`: This table stores information for Saga brokers. Rows are inserted into the `saga_message_broker$` table using an explicit `add_broker()` call. Brokers can be remote or local, and this information is captured using the `saga_message_broker$.remote` column.
- `sys.saga_participant$`: This table stores information for participants and coordinators of the Saga framework. Rows are inserted into the `saga_participant$` table using the `add_participant()` or the `add_coordinator()` calls. Participants can be remote or local, and this information is captured using the `saga_participant$.remote` column.
- `sys.Saga$`: The `sys.Saga$` table contains entries for Sagas, either initiated on the given PDB or joined (using `joinSaga()`).
- `sys.saga_finalization$`: The `saga_finalization$` table at the participant database records the sequence number and reservation journal information for each unique reservable table updated as part of the participant transaction. The `saga_finalization$` table does not maintain information for application-specific compensation.
- `sys.saga_participant_set$`: A Saga initiator sends AQ JMS messages to enroll Saga participants. The Saga AQ JMS messages use special JMS message properties to indicate the `saga_id` and other Saga attributes to the participants. An entry in `sys.saga_participant_set$` table tracks each participant enrolled in a Saga. These entries track the enrollment and finalization status for each participant.

- `sys.saga_pending$`: This table records the reservation journal compensation information for Sagas that have timed out. The Saga infrastructure uses this information to forcefully commit or rollback a Saga.
- `sys.saga_errors$`: This table records error messages corresponding to various Sagas.

The following two database parameters affect Sagas:

- `max_saga_duration`: This database parameter defines the maximum time in seconds that a Saga is considered to be active and is not marked incomplete. `max_saga_duration` is the system default for Saga duration, You can override this value using the `begin()` API. A Saga is marked incomplete if any of its participants are unable to finalize the Saga within the Saga duration. An incomplete Saga can be finalized using the `dbms_saga.rollback_saga()` interface.
- `saga_hist_retention`: This database parameter defines the maximum time in days for retaining information for completed Sagas. The default value for `saga_hist_retention` is 30 days.

28.6.7 Example: Saga Framework Setup

The following example configures a set of participant microservices and a broker.

This example sets up and configures a broker, two participants, namely TravelAgency and Airline, and their respective coordinator. Following these preparatory and provisioning steps, a Saga topology with two participants is created.

Preparatory Steps

Steps 1 through 7 are executed on `brokerPDB`, 8 through 11 at `TravelAgency`, and 12 through 15 at `AirlinePDB`.

1. Provision or designate a pluggable database (PDB) called `brokerPDB` to host the broker.
2. At `brokerPDB`, create a mailbox user that owns the broker and its JMS topic. For example: `MB`.
3. Grant the role `SAGA_ADM_ROLE` to user `MB`.
4. Create a proxy user `AirlineatMB` (Role: `SAGA_CONNECT_ROLE`).
5. Create a proxy user `TravelAgencyatMB` (Role: `SAGA_CONNECT_ROLE`).
6. Create dblink `LinkToAirline` to Airline PDB using `MBatAirline` schema.
7. Create dblink `LinkToTravelAgency` to Travel PDB using `MBatTravel` schema.
8. Provision or designate a PDB called `TravelAgency` to host Travel Reservation service.
9. At the `TravelAgency` PDB, create a user `TA`.
10. Create a proxy user `MBatTravelAgency` (Role: `SAGA_CONNECT_ROLE`).
11. Create dblink `LinktoBroker` to `brokerPDB` using `TravelAgencyatMB` schema.
12. Provision or designate a PDB called `AirlinePDB` to host the Airline.
13. At `AirlinePDB`, create user `Airline`.
14. Create proxy user `MBatAirline` (Role `SAGA_CONNECT_ROLE`).
15. Create dblink `LinktoBroker` to `brokerPDB` using `AirlineatMB` schema.

DBA views and entries associated with the dictionary entries in this example can be found in the DBA_SAGA_PARTICIPANTS view. For more information about the DBA_SAGA_PARTICIPANTS view and the following DBA views, see the Database Reference Guide.

- DBA_SAGAS
- DBA_HIST_SAGAS
- DBA_INCOMPLETE_SAGAS
- DBA_SAGA_DETAILS
- DBA_PARTICIPANT_SET
- DBA_SAGA_FINALIZATION
- DBA_SAGA_PENDING
- DBA_SAGA_ERRORS

Provisioning Steps

BrokerPDB:

```
--Add a broker
dbms_saga_adm.add_broker(name=>'TravelBroker',  schema=>'MB');
```

TravelAgencyPDB:

```
--Add the Saga coordinator(local to the initiator)
 dbms_saga_adm.add_coordinator(
 coordinator_name => 'TACoordinator',
 dblink_to_broker => 'LinktoBroker',
 mailbox_schema => 'MB',
 broker_name => 'TravelBroker',
 dblink_to_coordinator => 'LinkToTravelAgency'
);

--Add the local Saga participant TravelAgency and its coordinator as below
 dbms_saga_adm.add_participant(
 participant_name=> 'TravelAgency',
 coordinator_name=> 'TACoordinator',
 dblink_to_broker=> 'LinktoBroker',
 mailbox_schema=> 'MB',
 broker_name=> 'TravelBroker',
 dblink_to_participant=> 'LinktoTravelAgency',
 callback_package => 'dbms_ta_cbk'
);
```

AirlinePDB:

```
--Add the local Saga participant Airline as below
 dbms_saga_adm.add_participant(
 participant_name=> 'Airline',
 dblink_to_broker=> 'LinktoBroker',
 mailbox_schema=> 'MB',
 broker_name=> 'TravelBroker',
 dblink_to_participant=> 'LinktoAirline'
 callback_package => 'dbms_airline_cbk'
);

--Add a table with reservable column which maintains flight information
Create table flights(id NUMBER primary key,
 seats NUMBER reservable constraint flights_const check (seats > 0));
```

28.7 Managing a Saga Using the PL/SQL Interface

Use PL/SQL to develop packaged microservice applications in the database without requiring a mid-tier.

The PL/SQL package `DBMS_SAGA` enables you to use PL/SQL to develop packaged microservice applications in the database without requiring a mid-tier to communicate with a database. The `DBMS_SAGA` package provides the PL/SQL interfaces that enable client programs to initiate and interact with database Sagas.

> **See Also:**
>
> - DBMS_SAGA for the full description of the `DBMS_SAGA` package APIs
> - Developing Java Applications Using Saga Annotations

28.7.1 Example: Saga PL/SQL Program

The following is a PL/SQL sample program depicting a Saga initiator (TravelAgency) and a Saga participant (Airline).

Example 28-1 Saga Initiator

```
declare
  saga_id RAW(16);
  request JSON;
begin
  saga_id := dbms_saga.begin_saga('TravelAgency');
  request := json('{"flight":"United"}');
  dbms_saga.send_request(saga_id, 'Airline', request);
end;
/
```

Example 28-2 Airline

```
create or replace package dbms_airline_cbk as
function request(saga_id in RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)
return JSON;
end dbms_airline_cbk;
/

create or replace package body dbms_airline_cbk as
function request(saga_id in RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)
return JSON as
response JSON;
tickets NUMBER;
BEGIN
  BEGIN
    select seats into tickets from flights where id = json_value(payload, '$.flight')
for update;
    IF tickets > 0 THEN
      response := json('{"result":"success"}');
      update flights set seats = seats - 1 where id = json_value(payload, '$.flight');
    ELSE
      response := json('{"result":"failure"}');
```

```
    END IF;
  EXCEPTION
      WHEN OTHERS THEN
        response := json('{"result":"failure"}');
  END;
    return response;
end;

end dbms_airline_cbk;
/
```

Example 28-3 TravelAgency

```
create or replace package dbms_ta_cbk as
procedure response(saga_id in RAW, saga_sender IN varchar2, payload IN JSON DEFAULT
NULL);
end dbms_ta_cbk;
/

create or replace package body dbms_ta_cbk as

procedure response(saga_id in RAW, saga_sender IN varchar2, payload IN JSON DEFAULT
NULL) as
booking_result VARCHAR2(10);
begin
  booking_result := json_value(payload, '$.result');
  IF booking_result = 'success' THEN
    dbms_saga.commit_saga('TRAVELAGENCY', saga_id);
  ELSE
    dbms_saga.rollback_saga('TRAVELAGENCY', saga_id);
  END IF;
end;
end dbms_ta_cbk;
/
```

28.8 Developing Java Applications Using Saga Annotations

Java applications initiating and participating in Sagas should use the Saga annotations for flexibility and simplicity of development. In this section, you can find further details on Saga and LRA annotations that Java applications can leverage.

> **Note:**
>
> The Saga topology must be created separately.

> **See Also:**
>
> Setting Up a Saga Topology for more information about creating a Saga topology

The following participant types are used with Saga annotations.

Saga Initiator

A Saga initiator is a special Saga participant that is responsible for the Saga lifecycle. The initiator initiates a Saga, invites other participants to join the Saga, and finalizes the Saga. Only the initiator can send messages requesting other participants to join a Saga. A Saga initiator is associated with a Saga coordinator. Saga initiators use the standard `@LRA` annotation defined by the MicroProfile LRA specification that allows JAX-RS applications to initiate Sagas. The Saga framework's support for `@LRA`, `@Complete`, and `@Compensate` annotations is similar to the corresponding LRA annotation in the MicroProfile LRA specification. Instead of an LRA ID, the Saga framework's `@LRA` annotation initializes and returns a Saga ID to the application. The Java class for initiator must extend the `SagaInitiator` class provided by this framework.

> **See Also:**
>
> The `SagaInitiator` class in the section titled "Saga Interfaces and Classes"

Saga Participant

A Saga participant joins a Saga at the request of a Saga initiator. A participant can directly communicate with the initiator but not with other participants. Saga participants use the `@Request` annotation to mark the method invoked for processing incoming Saga payloads. Such methods consume a `SagaMessageContext` object that contains all the required metadata and payload for the Saga. The `@Request` annotated method returns a JSON object that the Saga framework automatically sends to the Saga initiator in response. A participant class must extend `SagaParticipant`.

> **See Also:**
>
> The `SagaParticipant` class and the `SagaMessageContext` class in the section titled "Saga Interfaces and Classes"

28.8.1 LRA and Saga Annotations

The Saga framework emulates the LRA framework and provides equivalent functionality. The following lists the annotations used in the Saga framework.

Table 28-1 LRA and Saga Annotations

Annotation	Description
`@LRA`	For Sagas, the LRA annotation controls the Saga initiation behavior. Currently, Sagas are handled using only an asynchronous model. The use of `end=true` is currently not supported. Generally, Saga finalization should be handled by calling `Saga.commitSaga()` or `Saga.rollbackSaga()` from a method that is annotated with `@Response`. The Saga implementation supports different LRA values, according to the LRA specification, except for annotation `@LRA.type.NESTED`. The Saga JAX-RS filter initializes a new Saga on behalf of the participant (described using the `@Participant` annotation), when needed. The `Saga ID` for the newly created Saga is inserted into the header parameter "Long-Running-Action" as defined by the specification. The method annotated with `@LRA` should be implemented by a class that extends the `SagaInitiator` class. This allows automatic instantiation of the Saga object as described by the `@LRA` annotation and the subsequent insertion of the LRA ID into the HTTP header. Participants can be invited to join a Saga using the `sendRequest()` method of the Saga class. For more information, see Saga Interface methods. Even if the initiator invokes `sendRequest()` to a participant multiple times for the same Saga, the participant joins the Saga only once. Currently, the Saga framework only supports `@LRA` on JAX-RS endpoints (using the JAX-RS filter) on the initiator. You can use the `beginSaga()` API to initiate Sagas manually.

Table 28-1 (Cont.) LRA and Saga Annotations

Annotation	Description

`@SagaConnection`	Any method annotated with `@SagaConnection` is used to fetch a JDBC Connection to create Saga Producers or Consumers while instantiating the Saga Participant.
	Each invocation of the method annotated with `@SagaConnection` should yield a new instance of a JDBC Connection.
	
	Note:
	When `close()` is called on a participant, the connections borrowed by calling the method decorated with `@SagaConnection` are closed (a `connection.close()` is invoked). You can change this behavior by modifying the `closable()` field of `@SagaConnection`.
	
	`@SagaConnection(closable=true)`
	`public java.sql.Connection getConnection() {`
	`.....`
	`}`

`@Complete`	`@Complete` is an LRA annotation that indicates the method the Saga framework automatically invokes when a Saga is committed. The Saga framework provides a `SagaMessageContext` object to the annotated method as an input argument. For more information, see the `SagaMessageContext` class. The method signature for a `@Complete` annotated method should be similar to: `public void airlineComplete(SagaMessageContext info)` . See the note at the end of this section for additional method signatures. The `@Complete` annotated method should finalize the local transactional changes on behalf of the Saga and clear any Saga state it was holding for possible Saga compensation. The method should use the connection object available in the `SagaMessageContext` class to perform any database actions and must not explicitly commit or roll back the database transaction.

Table 28-1 (Cont.) LRA and Saga Annotations

Annotation	Description
`@Compensate`	`@Compensate` is an LRA annotation that indicates the method the Saga framework automatically invokes when a Saga is rolled back. The Saga framework provides a `SagaMessageContext` object to the annotated method as an input argument. For more information, see the `SagaMessageContext` class. The method signature for a `@Compensate` annotated method should be similar to: `public void airlineCompensate(SagaMessageContext info)` See the note at the end of this section for additional method signatures. The `@Compensate` annotated method should compensate the local transactions performed during the Saga and clear any Saga state. The method should use the connection object available in the `SagaMessageContext` class to perform any database actions. The method should not explicitly commit or roll back the database transaction.
`@Participant`	`@Participant` is a Saga-specific annotation to indicate the method that maps a class to a Saga participant. The Saga participant must be previously defined within the database using the `dbms_saga_adm.add_participant()` API. The name of the participant also is used in the property file to associate additional parameters, such as the number of listeners and the number of publishers. **Note:** If `@SagaConnection` uses variables that require instantiation of the class decorated with `@SagaParticipant`, the `delay()` field of `@SagaParticipant` should be set to `true`. This delays the creation of Saga Producers and Consumers until the `start()` method is called on the class decorated with `@SagaParticipant`.
`@Request`	`@Request` is a Saga-specific annotation to indicate the method that receives incoming requests from Saga initiators. The Saga framework provides a `SagaMessageContext` object as an input to the annotated method. If the participant is working with multiple initiators, an optional sender attribute can be specified (regular expressions are allowed) to differentiate between them.
`@Response`	`@Response` is a Saga-specific annotation to indicate the method that collects responses from Saga participants enrolled into a Saga using the `sendRequest()` API. The Saga framework provides a `SagaMessageContext` object as an input to the annotated method. If the initiator is working with multiple participants, an optional sender attribute can be specified (regular expressions are allowed) to differentiate between them.

Table 28-1 (Cont.) LRA and Saga Annotations

Annotation	Description
`@BeforeComplete`	`@BeforeComplete` is a Saga-specific annotation to indicate the method that is invoked during Saga finalization before a Saga is committed. The method annotated with `@BeforeComplete` is invoked before the automatic completion for any lock-free reservations performed by the Saga. The use of `@BeforeComplete` is optional.
`@BeforeCompensate`	`@BeforeCompensate` is a Saga-specific annotation to indicate the method that is invoked during Saga finalization before a Saga is rolled back. The method annotated with `@BeforeCompensate` is invoked before the automatic compensation for any lock-free reservations performed by the Saga. The use of `@BeforeCompensate` is optional.
`@InviteToJoin`	`@InviteToJoin` is a Saga-specific annotation that indicates the method that is invoked when the initiator requests that a particular participant can join a given Saga (using the `sendRequest()` API). If the method returns true, the participant joins the Saga. Otherwise, a negative acknowledgment is returned, and `@Reject` is invoked. The use of `@InviteToJoin` is optional.
`@Reject`	`@Reject` is a Saga-specific annotation to indicate the method that is invoked when: • A participant declines to join a Saga as defined by `@InviteToJoin`. • An unhandled exception is raised in the participant, in the method indicated by `@Request`. `@Reject` annotation is applicable only for an initiator. The method annotated with `@Reject` can perform bookkeeping for the Saga at the initiator level.

Note:

For all annotations with an associated method, three method signatures are supported. For example, consider the following method signatures for the `@Request` annotation.

- `ProcessPayload (SagaMessageContext info)`
- `ProcessPayload (URI sagaId)`
- `ProcessPayload (URI sagaId, URI parentId)`

Since the Saga framework does not support `NESTED` Sagas, `parentId` is always null. For using these alternate signatures, a utility method: `getSagaMessageContext()` is provided to retrieve the `SagaMessageContext` object from a given Saga ID.

> **Note:**
>
> - All database operations should be performed using the database connection object found in the `SagaMessageContext` class, and an explicit transaction commit or rollback is not supported.
> - If there are multiple matches on the sender value, an exception is raised.

28.8.2 Packaging

The Saga framework comprises two libraries:

- The Saga Client library
- The JAX-RS filter

The JAX-RS filter is only needed if the application is a JAX-RS application and wishes to initiate Sagas using the `@LRA` annotation.

The following are the relevant Maven coordinates for the two libraries:

Saga Client Library

```
<dependency>
  <groupId>com.oracle.database.saga</groupId>
  <artifactId>saga-core</artifactId>
  <version>${SAGA_VERSION}</version>
</dependency>
```

JAX-RS Filter

```
<dependency>
  <groupId>com.oracle.database.saga</groupId>
  <artifactId>saga-filter</artifactId>
  <version>${SAGA_VERSION}</version>
</dependency>
```

28.8.3 Configuration

JAX-RS Filter Configuration

The JAX-RS filter has the following parameters that can be configured using a property file (`sagafilter.properties`, `application.properties`, or both):

Property Name	Description
`osaga.filter.database.tnsAlias`	The TNS alias to use from the `tnsnames.ora` file
`osaga.filter.database.walletPath`	The path to the wallet that needs to be used for the filter connection
`osaga.filter.database.tnsPath`	The path to the location of the `tnsnames.ora` file

Property Name	Description
`osaga.filter.initiator.publisherCount`	The total number of publishers to instantiate in the publisher pool that is used to initiate Sagas Each time a new Saga needs to be initiated, an existing publisher from the pool is used. Once the Saga is initiated, the publisher is released back into the pool.
`osaga.filter.initiator.name`	The `osaga.filter.initiator.name` property should match the `@Participant` annotation value as the filter acts on behalf of the initiator.

> **Note:**
>
> The Saga JAX-RS filter should be the only JAX-RS LRA filter in the environment. Multiple LRA filters can cause inconsistency and unpredictable results.

Participants Configuration

Each of the participants has the following parameters that can be configured using a property file (`osaga.app.properties`, `application.properties`, or both):

Property Name	Description
`osaga.<participant_name>.tnsAlias`	The TNS alias to use from the `tnsnames.ora` file **Note:** This parameter is not required if `@SagaConnection` is used.
`osaga.<participant_name>.walletPath`	The path to the wallet that needs to be used for the participant connection **Note:** This parameter is not required if `@SagaConnection` is used.

Property Name	Description
`osaga.<participant_name>.tnsPath`	The path to the location of the `tnsnames.ora` file **Note:** This parameter is not required if `@SagaConnection` is used.
`osaga.<participant_name>.numListeners`	The number of listeners assigned to this participant per queue partition. Listeners are used to process incoming messages. In practice, the number of listeners configured for a participant or an initiator depends on the number of other participants and initiators that can communicate with it. This number may have to be increased due to other factors, such as when the processing time for the payloads is higher or the number of concurrent requests increase.
`osaga.<participant_name>.numPublishers`	Publishers are needed to send the requests to participants. In practice, the number of publishers depends on the expected number of concurrent Sagas. The `<participant_name>` is derived from the `@Participant` annotation. For example, a class marked with `@Participant(name="TravelAgency")` is configured using the `osaga.travelagency` prefix, for example, `osaga.travelagency.numListeners=2`.

Note:

`numListeners` and `numPublishers` are independent entities. `numListeners` refers to the number of threads responding to incoming requests. `numPublishers` refers to an initiator's publisher threads. For an initiator, it is ideal to have `numListeners=numPublishers` for reliable throughput.

28.8.4 Saga Interface and Classes

The Saga interface and classes describe the functionality required to support Sagas. This section describes the interface and classes for Java applications.

28.8.4.1 Saga Interface

The Saga interface provides the means to participate in, complete, and request metadata for a database Saga. The Saga interface provides the following methods:

`sendRequest(String recipient, String payload)` Method

Syntax:

```
* @param recipient - name of the participant to be enrolled
* @param payload - saga payload
public void sendRequest(String recipient, String payload)
```

Description: The `sendRequest(String recipient,String payload)` method is invoked at the initiator level to send a message (payload) to a participant. If the participant joins the Saga, the initiator invokes the method annotated with `@Response`. Otherwise, the initiator invokes the method annotated with `@Reject`.

> **Note:**
>
> The `sendRequest(String recipient,String payload)` method automatically commits the transaction by default when used outside the scope of Saga-annotated methods. Fine-grained control over the transaction boundary can be achieved using the `sendRequest(java.sql.Connection connection, String recipient, String payload)` method.

`sendRequest(AQjmsSession session, TopicPublisher publisher, String recipient, String payload)` Method

> **Note:**
>
> This method is deprecated. Instead, use the `sendRequest(java.sql.Connection connection, String recipient, String payload)` method.

Syntax:

```
* @param session - user supplied saga session
* @param publisher - user supplied topic publisher
* @param recipient - name of the participant to be enrolled
* @param payload - saga payload
public void sendRequest(AQjmsSession session, TopicPublisher publisher,
String recipient, String payload)
```

Description: This method is identical to `sendRequest(String recipient,String payload)` except for the use of `AQjmsSession` and `TopicPublisher` parameters.

Multiple `sendRequest()` calls can be made in a single database transaction, provided they have identical `AQjmsSession` session and `TopicPublisher` publisher parameter values. Other database operations (such as DML) can also be part of the same transaction, provided they use the database connection embedded in `AQjmsSession` session obtained using the `getDBConnection()` method of the `AQjmsSession` class. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `AQjmsSession`.

> **Note:**
>
> A `TopicPublisher` (publisher) instance is obtained using the `getSagaOutTopicPublisher` (`AQjmsSession` session) method declared in the `SagaInitiator` class.

`sendRequest(java.sql.Connection connection, String publisher, String recipient, String payload)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
* @param publisher - user supplied topic publisher
* @param recipient - name of the participant to be enrolled
* @param payload - saga payload
public void sendRequest(java.sql.Connection connection, String publisher,
String recipient, String payload)
```

Description: This method is identical to `sendRequest(String recipient,String payload)` except for the use of `java.sql.Connection connection` and the `publisher` parameter.

Multiple `sendRequest()` calls can be made in a single database transaction, provided they use the supplied `java.sql.Connection` connection. Other database operations (such as DML) can also be part of the same transaction, provided they use the supplied database connection. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `Connection`.

`getSagaId()` Method

Syntax:

```
public String getSagaId()
```

Description: The `getSagaId()` method returns the Saga identifier associated with the Saga object instantiated using `beginSaga()` or `getSaga(String sagaId)` method in the `SagaInitiator` class.

`commitSaga()` Method

Syntax:

```
public void commitSaga()
```

Description: The `commitSaga()` method is invoked by the initiator to commit the Saga. As part of its execution, the methods annotated with `@BeforeComplete` and `@Complete` are invoked at both the initiator and participants levels. Reservable column operations, if any, are finalized between the `@BeforeComplete` and `@Complete` calls.

> **Note:**
>
> The `commitSaga()` method auto commits the transaction by default, when used outside the scope of Saga-annotated methods. Fine-grained control over the transaction boundary can be achieved using the `commitSaga(java.sql.Connection connection)` method.

`commitSaga(AQjmsSession session)` Method

> **Note:**
>
> This method is deprecated. Instead, use the `commitSaga(java.sql.Connection connection)` method.

Syntax:

```
* @param session - user supplied saga session
public void commitSaga(AQjmsSession session)
```

Description: This method is identical to the `commitSaga()` method except for the use of `AQjmsSession` session.

Other database operations (such as DML) can also be part of the same transaction, provided they use the database connection embedded in the `AQjmsSession` session obtained using the `getDBConnection()` method of the `AQjmsSession` class. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `AQjmsSession`.

`commitSaga(java.sql.Connection connection)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
public void commitSaga(java.sql.Connection connection)
```

Description: This method is identical to the `commitSaga()` method except for the use of `java.sql.Connection` connection.

Other database operations (such as DML) can also be a part of the same transaction, provided they use the supplied database connection. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `Connection`.

`commitSaga(boolean force)` Method

Syntax:

```
* @param force - force commit flag
public void commitSaga(boolean force)
```

Description: This method is identical to `commitSaga()` except for the use of the `force` flag.

The `force` flag could be used by a Saga participant in special situations to locally commit the Saga and inform the Saga coordinator without waiting for the finalization from the Saga initiator.

`commitSaga(AQjmsSession session, boolean force)` Method

> **Note:**
>
> This method is deprecated. Instead, use the `commitSaga(java.sql.Connection connection, boolean force)` method.

Syntax:

```
* @param session - user supplied saga session
* @param force - force commit flag
public void commitSaga(AQjmsSession session, boolean force)
```

Description: This method is identical to the `commitSaga()` method except for the use of the `AQjmsSession` session and the `force` flag. It combines the functionality of the `commitSaga(AQjmsSession session)` and `commitSaga(boolean force)` methods.

> **Note:**
>
> The `commitSaga(AQjmsSession session)`, `commitSaga(boolean force)`, and `commitSaga(AQjmsSession session, boolean force)` methods can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction, by default, which is committed or rolled back after the execution of the method.

`commitSaga(java.sql.Connection connection, boolean force)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
* @param force - force commit flag
public void commitSaga(java.sql.Connection connection, boolean force)
```

Description: This method is identical to the `commitSaga()` method except for the use of the `java.sql.Connection` connection and the `force` flag. It combines the functionality of the `commitSaga(java.sql.Connection connection)` and `commitSaga(boolean force)` methods.

> **Note:**
>
> The `commitSaga(java.sql.Connection connection)`, `commitSaga(boolean force)`, and `commitSaga(java.sql.Connection connection, boolean force)` methods can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction, by default, which is committed or rolled back after the execution of the method.

`rollbackSaga()` Method

Syntax:

```
public void rollbackSaga()
```

Description: The `rollbackSaga()` method rolls back the Saga and invokes the methods annotated with `@BeforeCompensate` and `@Compensate` annotations. Reservable column operations (if any) are finalized between the `@BeforeCompensate` and `@Compensate` calls.

> **Note:**
>
> The `rollbackSaga()` method auto commits the transaction by default if used outside the scope of Saga-annotated methods. Fine-grained control over the transaction boundary can be achieved using the `rollbackSaga(java.sql.Connection connection)` method.

`rollbackSaga(AQjmsSession session)` Method

> **Note:**
>
> This method is deprecated. Instead, use the `rollbackSaga(java.sql.Connection connection)` method.

Syntax:

```
* @param session - user supplied saga session
public void rollbackSaga(AQjmsSession session)
```

Description: This method is identical to the `rollbackSaga()` method except for the use of the `AQjmsSession` session.

Other database operations (such as DML) can also be part of the same transaction provided they use the database connection embedded in the `AQjmsSession` session obtained using the `getDBConnection()` method of the `AQjmsSession` class. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `AQjmsSession`.

> **Note:**
>
> The `rollbackSaga(AQjmsSession session)` method can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction by default which is committed or rolled back after execution of the method.

`rollbackSaga(java.sql.Connection connection)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
public void rollbackSaga(java.sql.Connection connection)
```

Description: This method is identical to the `rollbackSaga()` method except for the use of the `java.sql.Connection` connection.

Other database operations (such as DML) can also be part of the same transaction provided they use the supplied database connection. The transaction can be committed or rolled back using the `commit()` or `rollback()` method in `Connection`.

> **Note:**
>
> The `rollbackSaga(java.sql.Connection connection)` method can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction by default which is committed or rolled back after execution of the method.

`rollbackSaga(boolean force)` Method

Syntax:

```
* @param force - force rollback flag
public void rollbackSaga(boolean force)
```

Description: This method is identical to the `rollbackSaga()` method except for the use of the `force` flag.

The `force` flag could be used by a Saga participant in special situations to locally roll back the Saga and inform the Saga coordinator without waiting for the finalization from the Saga initiator.

`rollbackSaga(AQjmsSession session, boolean force)` Method

> **Note:**
>
> This method is deprecated. Instead, use the `rollbackSaga(java.sql.Connection connection, boolean force)` method.

Syntax:

```
* @param session - user supplied saga session
* @param force - force commit flag
public void rollbackSaga(AQjmsSession session, boolean force)
```

Description: This method is identical to the `rollbackSaga()` method except for the use of the `AQjmsSession` session and the `force` flag. It combines the functionality of the `rollbackSaga(AQjmsSession session)` and `rollbackSaga(boolean force)` methods.

> **Note:**
>
> The `rollbackSaga(AQjmsSession session)`, `rollbackSaga(boolean force)`, and `rollbackSaga(AQjmsSession session, boolean force)` can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction, by default, which is committed or rolled back after execution of the method.

`rollbackSaga(java.sql.Connection connection, boolean force)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
* @param force - force commit flag
public void rollbackSaga(java.sql.Connection connection, boolean force)
```

Description: This method is identical to the `rollbackSaga()` method except for the use of the `java.sql.Connection` connection and the `force` flag. It combines the functionality of the `rollbackSaga(java.sql.Connection connection)` and `rollbackSaga(boolean force)` methods.

> **Note:**
>
> The `rollbackSaga(java.sql.Connection connection)`, `rollbackSaga(boolean force)`, and `rollbackSaga(java.sql.Connection connection, boolean force)` can only be used outside the scope of Saga-annotated methods. Saga-annotated methods begin a new transaction, by default, which is committed or rolled back after execution of the method.

`isSagaFinalized()` Method

Syntax:

```
public boolean isSagaFinalized()
```

Description: The `isSagaFinalized()` method can be invoked at an initiator or participant level and it checks whether the Saga has reached one of the finalization states. It returns false if the saga is in a `JOINING`, `JOINED`, or `TIMEDOUT` state, and returns true, otherwise.

`beginSagaTransaction(AQjmsSession session, TopicPublisher publisher)` Method

> **Note:**
>
> This method is deprecated. Instead, use the `beginSagaTransaction(java.sql.Connection connection)` method.

Syntax:

```
* @param session - user supplied saga session
* @param publisher - user supplied topic publisher
public void beginSagaTransaction(AQjmsSession session, TopicPublisher
publisher)
```

Description: The `beginSagaTransaction(AQjmsSession session, TopicPublisher publisher)` method is used outside the scope of a Saga-annotated method (for example, under the `@LRA annotated` method) to start a new Saga transaction. It can only be invoked at an initiator level. All operations performed inside the Saga transaction are committed or rolled back explicitly by calling the `commit()` or `rollback()` method of `AQjmsSession`. The `beginSagaTransaction(AQjmsSession session, TopicPublisher publisher)` call requires a corresponding `endSagaTransaction()` call to complete the Saga transaction.

> **Note:**
>
> The `beginSagaTransaction(AQjmsSession session, TopicPublisher publisher)` method is serialized with the invocation of other Saga related methods, such as `commitSaga()` or `rollbackSaga()`.

`beginSagaTransaction(java.sql.Connection connection, String publisher)` Method

Syntax:

```
* @param connection - user supplied JDBC connection
* @param publisher - user supplied topic publisher
public void beginSagaTransaction(java.sql.Connection connection, String
publisher)
```

Description: The `beginSagaTransaction(java.sql.Connection connection)` method is used outside the scope of a Saga-annotated method (for example, under the `@LRA annotated` method) to start a new Saga transaction. It can only be invoked at an initiator level. All operations performed inside the Saga transaction are committed or rolled back explicitly by calling the `commit()` or `rollback()` method in `Connection`. The `beginSagaTransaction(java.sql.Connection connection)` call requires a corresponding `endSagaTransaction()` call to complete the Saga transaction.

> **Note:**
>
> The `beginSagaTransaction(java.sql.Connection connection)` method is serialized with the invocation of other Saga related methods, such as `commitSaga()` or `rollbackSaga()`.

`endSagaTransaction()` Method

Syntax:

```
public void endSagaTransaction()
```

Description: The `endSagaTransaction()` method ends the Saga transaction started by a `beginSagaTransaction()` call. The `endSagaTransaction()` method releases the lock upon a Saga if the Saga transaction was started with the lock flag set to true.

28.8.4.2 SagaMessageContext Class

The `SagaMessageContext` object is passed as an argument to methods annotated with Saga annotations upon a Saga related event. The `SagaMessageContext` object can be used to fetch metadata associated with the Saga that triggered the Saga annotated method.

The `SagaMessageContext` class provides the following methods.

`getSagaId()` Method

Syntax: `public String getSagaId()`

Description: The `getSagaId()` method returns the identifier of the Saga.

`getSender()` Method

Syntax: `public String getSender()`

Description: The `getSender()` method returns the name of the sender of the message.

`getPayload()` Method

Syntax: `public String getPayload()`

Description: The `getPayload()` method returns the payload associated with the Saga message.

`getConnection()` Method

Syntax: `public java.sql.Connection getConnection()`

Description: The `getConnection()` method can be used in any Saga-annotated method to return the database connection object of the Saga message listener thread. It can be used at an initiator or participant level. The connection can be used by the application to perform database operations.

Explicit `commit()` or `rollback()` in the saga-annotated method is not supported.

`getSaga()` Method

Syntax: `public Saga getSaga()`

Description: The `getSaga()` method returns the Saga object associated with the Saga that triggered the Saga annotated method. The Saga object can be used to finalize that Saga or request metadata for that Saga.

28.8.4.3 SagaParticipant Class

The `SagaParticipant` class provides the means to create and manage a Saga participant instance. The Saga participant is created using the `add_participant()` PL/SQL API provided in the `DBMS_SAGA_ADM` package.

> **See Also:**
>
> DBMS_SAGA_ADM for a complete description of the `SYS.DBMS_SAGA_ADM` package APIs.

The `SagaParticipant` class exposes the following methods.

`addSagaMessageListener()` Method

Syntax:

```
public void addSagaMessageListener()
```

Description: The `addSagaMessageListener()` method allows a Saga participant to add an additional message listener thread. Adding more Saga message listener threads allows the application to increase concurrency when processing Saga messages.

> **Note:**
>
> `addSagaMessageListener()` adds only one message listener thread. To add more than one listener thread at once, the `addSagaMessageListener(int numListeners)` method should be used.

`addSagaMessageListener(int numListeners)` Method

Syntax:

```
* @param numListeners - number of listeners to add
public void addSagaMessageListener(int numListeners)
```

Description: The `addSagaMessageListener(int numListeners)` method allows a Saga participant to add additional message listener threads as `numListeners`. Addition of more Saga message listener threads allows for processing more Saga messages concurrently.

`removeSagaMessageListener()` Method

Syntax:

```
public void removeSagaMessageListener()
```

Description: The `removeSagaMessageListener()` method allows a Saga participant to remove a message listener thread. Under lower load, Saga message listener threads can be reduced to free up system resources.

> **Note:**
>
> The `removeSagaMessageListener()` method removes only one message listener thread. To remove more than one listener thread at once the `removeSagaMessageListener (int numListeners)` method should be used.

`removeSagaMessageListener(int numListeners)` Method

Syntax:

```
* @param numListeners - number of listeners to remove
public void removeSagaMessageListener(int numListeners)
```

Description: The `removeSagaMessageListener(int numListeners)` method allows a Saga participant to remove `numListeners` message listener threads. Under lower load, Saga message listener threads can be reduced to free up system resources.

`removeAllSagaMessageListeners()` Method

Syntax:

```
public void removeAllSagaMessageListeners()
```

Description: The `removeAllSagaMessageListeners()` method allows a Saga participant to remove all Saga message listener threads. Invoking this method prevents the execution of the methods annotated with Saga-specific annotations upon receiving a Saga message. This method can be used where the business logic of the methods annotated with Saga annotations needs to be changed. Any unprocessed Saga messages are retained in the Saga message queues and can be processed once message listeners restart.

`getSagaMessageListenerCount()` Method

Syntax:

```
public int getSagaMessageListenerCount()
```

Description: The `getSagaMessageListenerCount()` method returns the total number of Saga message listener threads associated with a Saga participant.

> **Note:**
>
> In case of an AQ queue setup, `getSagaMessageListenerCount()` returns the message listener threads associated with a single partition of the Saga participant. Total message listener threads are obtained by `getSagaMessageListenerCount() * numPartitions`, which are defined during the `add_participant` call. The number of partitions configured for a participant can also be queried using the `DBA_SAGA_PARTITIPANTS` dictionary view.

`getSaga(String sagaId)` Method

Syntax:

```
* @param sagaId - Identifier for the saga to be retrieved
public Saga getSaga(String sagaId)
```

Description: The `getSaga(String sagaId)` method returns the Saga object associated with the Saga with the specified Saga identifier.

`close()` Method

Syntax:

```
public void close()
```

Description: The `close()` method removes all message listener threads and connections associated with a Saga participant. When the Saga participant is closed, methods annotated with Saga annotations are not invoked any further. The Saga participant cannot participant in, complete, and request metadata for a database Saga. The pending Saga messages for the participant are retained and can be consumed in the future.

28.8.4.4 SagaInitiator Class

The `SagaInitiator` class provides the means to create and manage a Saga initiator instance. The Saga initiator is created using the `add_participant()` PL/SQL API provided in the `dbms_saga_adm` package. The `SagaInitiator` class inherits all methods specified in the `SagaParticipant` class. Additionally, the `SagaInititor` class exposes the following methods.

`beginSaga()` Method

Syntax:

```
public Saga beginSaga()
```

Description: The `beginSaga()` method starts a Saga and returns a `sagaId`. The `sagaId` can be used to enroll other participants.

> **Note:**
>
> The `beginSaga()` method starts a Saga with the default timeout of 84600 seconds. Fine-grained control over the timeout can be achieved using `beginSaga(int timeout)`.

`beginSaga(int timeout)` Method

Syntax:

```
* @param timeout - saga timeout
public Saga beginSaga(int timeout)
```

Description: The `beginSaga(int timeout)` method starts a Saga and returns a Saga identifier with a user-specified timeout. The Saga identifier can be used to enroll other participants. Note: If the Saga is not finalized before the specified timeout, the Saga automatically finalizes depending on the database initialization parameter `_saga_timeout_operation_type(default=rollback)`.

`addSagaMessagePublishers(int numPublishers)` Method

Syntax:

```
* @param numPublisher - number of publishers to add
public void addSagaMessagePublishers(int numPublishers)
```

Description: The `addSagaMessagePublishers(int numPublishers)` method allows a Saga initiator to add additional message publisher threads as `numPublisher`. Addition of more Saga message publisher threads allows the initiator to create and manage more Sagas concurrently.

`removeSagaMessagePublishers(int numPublishers)` Method

Syntax:

```
* @param numPublisher - number of publishers to remove
public void removeSagaMessagePublishers(int numPublishers)
```

Description: The `removeSagaMessagePublishers(int numPublishers)` method allows a Saga initiator to remove `numPublishers` message publisher threads. Under lower load, Saga message publisher threads can be reduced to free up system resources.

`removeAllSagaMessagePublishers()` Method

Syntax:

```
public void removeAllSagaMessagePublishers()
```

Description: The `removeAllSagaMessagePublishers()` method allows a Saga initiator to remove all Saga message publisher threads. All message publisher threads can be removed if the initiator does not want to initiate or manage any Sagas further. Removing Saga message publishers does not affect the pending messages previously published.

`getSagaMessagePublisherCount()` Method

Syntax:

```
public int getSagaMessagePublisherCount()
```

Description: The `getSagaMessagePublisherCount()` method return the total number of saga message publisher threads associated with a saga initiator.

> **Note:**
>
> In case of a AQ queue setup, the `getSagaMessagePublisherCount()` method returns the message publisher threads associated with a single partition of the Saga participant. Total message publisher threads are obtained by `getSagaMessagePublisherCount() * numPartitions`, which are defined during the `add_participant()` call. The number of partitions configured for a participant can also be queried using the `DBA_SAGA_PARTITIPANTS` dictionary view.

`getSagaOutTopicPublisher(AQjmsSession session, int partition)` Method

Syntax:

```
* @param session - user supplied saga session
* @param partition - partition for which the publisher is needed
public TopicPublisher getSagaOutTopicPublisher(AQjmsSession session, int
partition)
```

Description: The `getSagaOutTopicPublisher(AQjmsSession session, int partition)` method returns the Saga topic publisher instance. The `TopicPublisher` instance can be supplied as a parameter to various methods that accept a user defined `AQjmsSession` to manage the transaction boundary manually.

> **Note:**
>
> For AQ queue setup, the value of partition could be `[1-numPartitions]` specified in the `add_participant` call. In case of a TxEventQ queue setup, the value of partition should always be 1.

28.8.5 Example Program

The following example illustrates the use of a subset of the important Saga annotations through a simple Travel Agency Saga application. The Travel Agency emulates a real travel agency that performs Airline reservations for its end users. The Travel Agency is a JAX-RS application (Initiator) that uses one participant: Airline. Airline is implemented as a standard Java application (not JAX-RS).

The Travel Agency and Airline leverage appropriate Saga annotations for performing their tasks. In this example, `TravelAgency` is the Saga initiator that initiates a Saga to purchase

airline tickets for its end users. `Airline` is the Saga participant. The example is a simple illustration where only one reservation is made by a participant towards a Saga. In practice, a Saga may span multiple participants.

> **Note:**
>
> An application developer only needs to implement the annotated methods as shown in this example. The Saga framework provides the necessary communication and maintains the state of the Saga across a distributed topology.

> **Note:**
>
> The `@LRA`, `@Compensate`, `@Complete` annotations are LRA annotations, whereas `@participant`, `@Request`, and `@Response` are Saga annotations.

Example 28-4 TravelAgency (Saga Initiator)

```
@Participant(name = "TravelAgency")
/* @Participant declares the participant's name to the saga framework */
public class TravelAgencyController extends SagaInitiator {
/* TravelAgencyController extends the SagaInitiator class */
 @LRA(end = false)
 /* @LRA annotates the method that begins a saga and invites participants */
 @POST("booking")
 @Consumes(MediaType.TEXT_PLAIN)
 @Produces(MediaType.APPLICATION_JSON)
  public jakarta.ws.rs.core.Response booking(
      @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
      String bookingPayload) {
    Saga saga = this.getSaga(lraId.toString());
    /* The application can access the sagaId via the HTTP header
       and instantiate the Saga object using it */
    try {
      /* The TravelAgency sends a request to the Airline sending
         a JSON payload using the Saga.sendRequest() method */
      saga.sendRequest ("Airline", bookingPayload);
      response = Response.status(Response.Status.ACCEPTED).build();
    } catch (SagaException e) {
      response=Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
    }
  }
  @Response(sender = "Airline.*")
  /* @Response annotates the method to receive responses from a specific
     Saga participant */
  public void responseFromAirline(SagaMessageContext info) {
    if (info.getPayload().equals("success")) {
      saga.commitSaga ();
      /* The TravelAgency commits the saga if a successful response is
received */
    } else {
      /* Otherwise, the TravelAgency performs a Saga rollback  */
```

```
      saga.rollbackSaga ();
    }
  }
}
```

Example 28-5 Airline (Saga Participant)

```
@Participant(name = "Airline")
/* @Participant declares the participant's name to the saga framework */
public class Airline extends SagaParticipant {
/* Airline extends the SagaParticipant class */
  @Request(sender = "TravelAgency")
  /* The @Request annotates the method that handles incoming request from a
given
     sender, in this example the TravelAgency */
  public String handleTravelAgencyRequest(SagaMessageContext
      info) {

    /* Perform all DML with this connection to ensure
       everything is in a single transaction */
    FlightService fs = new
      FlightService(info.getConnection());
    fs.bookFlight(info.getPayload(), info.getSagaId());
    return response;
    /* Local commit is automatically performed by the saga framework.
       The response is returned to the initiator */
  }

  @Compensate
  /* @Compensate annotates the method automatically called to roll back a
saga */
  public void compensate(SagaMessageContext info) {
    fs.deleteBooking(info.getPayload(),
        info.getSagaId());
  }
  @Complete
  /* @Complete annotates the method automatically called to commit a saga */
  public void complete(SagaMessageContext info) {
    fs.sendConfirmation(info.getSagaId());
  }
}
```

> **Note:**
>
> The import directives are not included in the sample program.

Other microservices, such as `Hotel` and `CarRental` would have a structure similar to the Airline microservice.

The Travel Agency could be modified in the following ways to accommodate another participant. Consider adding a Car Rental service:

- In TravelAgencyController's booking method, another `sendRequest()` is required to send the payload to the "Car" participant.

```
Saga.sendRequest ("Car", bookingPayload.getCar().toString());
```

- A new annotated method is required to handle responses from the Car service.

```
@Response(sender = "Car”)
```

- Additional logic in the `@Response` methods is necessary to account for the state of the reservation after messages are received from participants.

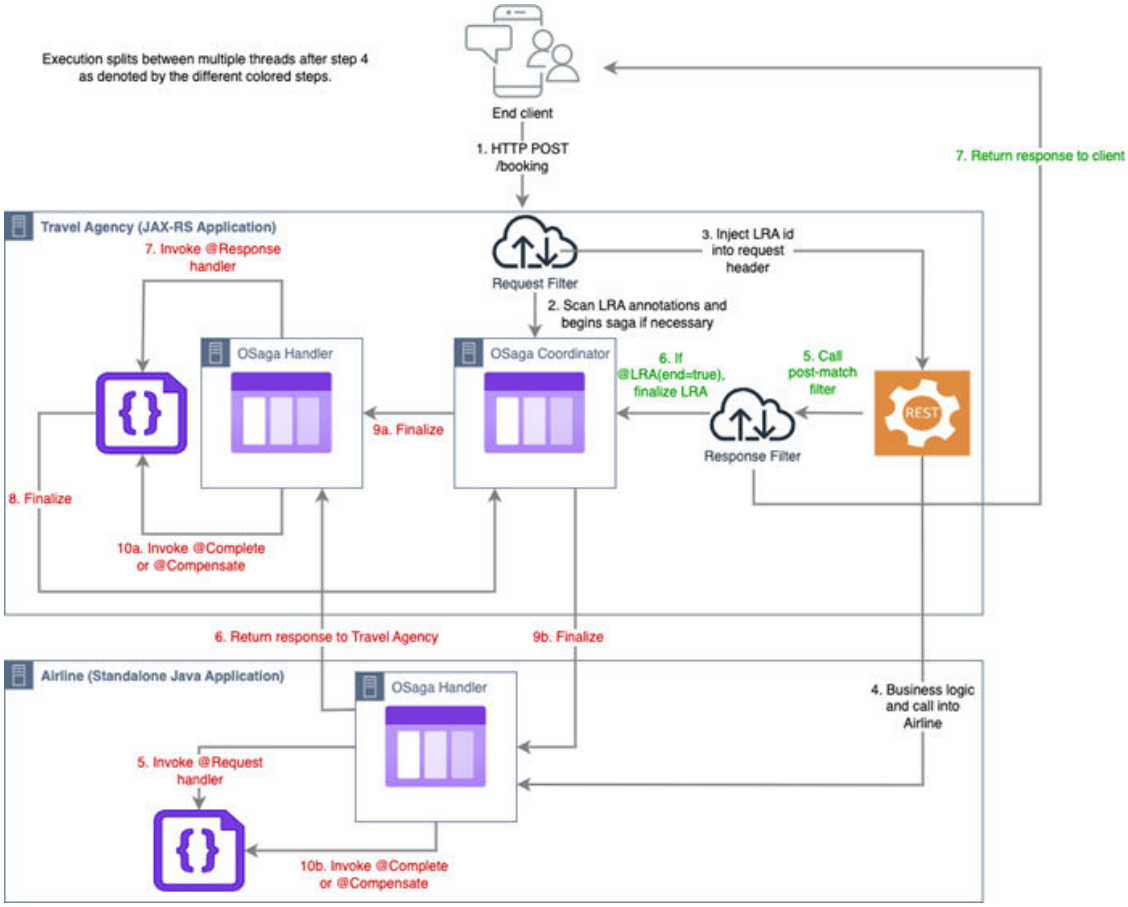
```
@Response(sender = "Car")
public void responseFromCar(SagaMessageContext info) {
  if (!info.getPayload().equals("success")) {
    /* On error, rollback right away */
    Saga.rollbackSaga ();
  } else {
    /* Store the car's response */
    cache.putCarResponse(info.getSagaId(), info.getPayload());

    /* Check to see if Airline has responded
       If a response is found, commit */
    if (cache.containsAirlineResponse(info.getSagaId())) {
      Saga.commitSaga ();
    }
  }
}
```

The method to handle Airline's response would be similar, except that it would replace `cache.containsAirlineResponse(info.getSagaId())` with `cache.containsCarResponse(info.getSagaId())` and `cache.putAirlineResponse()` with `cache.putCarResponse()`.

The following asynchronous flow diagram shows the process flow of a Saga application illustrated in the previous code examples. The numbers in the flow diagram correspond to the numbers shown in the code snippet.

> **See Also:**
>
> Appendix: Troubleshooting the Saga Framework for more information about the Saga lifecycle

Figure 28-1 Asynchronous Flow



Saga Property File

The following shows the Saga property file used by the example program.

```
# JAX-RS filter properties
osaga.filter.database.tnsAlias=ta
osaga.filter.database.walletPath=/Path/to/wallet
osaga.filter.database.tnsPath=/Path/to/tns
osaga.filter.initiator.publisherCount=2
osaga.filter.initiator.name=TravelAgency
osaga.travelagency.tnsAlias=ta

# Property values for the Initiator Travelagency
osaga.travelagency.walletPath=/Path/to/wallet
osaga.travelagency.tnsPath=/Path/to/tns
osaga.travelagency.numListeners=2
osaga.travelagency.numPublishers=2

# Properties pertaining to the Airline participant
osaga.airline.tnsAlias=airline1
osaga.airline.walletPath=/Path/to/wallet
osaga.airline.tnsPath=/Path/to/tns
osaga.airline.numListeners=2
osaga.airline.numPublishers=2
```

28.9 Finalizing a Saga Explicitly

In addition to the automatic reservable column-based finalization, the Saga framework supports finalization (complete or compensate) that is application-specific and explicit (user-defined). The framework automatically invokes application-specific finalization routines during Saga finalization.

You can designate the application-specific finalization using the PL/SQL callback routines for the PL/SQL clients.

> **Note:**
>
> Ensure that there is no explicit `COMMIT` command issued from the user-defined callbacks that are implemented using the PL/SQL callback package. The Saga framework commits the changes on behalf of the application.

For Java programs, Developing Java Applications Using Saga Annotations describes the `@BeforeComplete`, `@BeforeCompensate`, and `@Complete`, `@Compensate` annotations that a Java program can use to implement application-specific compensation. In such cases, the lock-free reservations are automatically compensated between the `@BeforeCompensate` and `@Compensate` annotated methods.

> **See Also:**
>
> - Using Lock-Free Reservation
> - Integration with Lock-Free Reservation

28.9.1 PL/SQL Callbacks for a PL/SQL Client

A PL/SQL based initiator or participant needs to implement a callback package to participate in Sagas. The callback package is called internally from the notification callback for the respective participant. The callback package enables you to ignore the internal details of the Saga infrastructure and focus on your business logic.

Applications must ensure that the implementation of a callback package does not invoke any database transaction control operations (commit/rollback).

The callback package provides the following functions:

`request` Function

Syntax: `function request(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL) return JSON;`

Description: The request function is invoked when receiving a Saga message with opcode `REQUEST`. The application is expected to implement this method. A JSON payload is returned that is sent back to the initiator as a response.

`response` Procedure

Syntax: `procedure response(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The response procedure is invoked when receiving a Saga message with opcode `RESPONSE`. The application is expected to implement this method.

`before_commit` Procedure

Syntax: `procedure before_commit(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The `before_commit` procedure is invoked before executing the Saga commit on the current participant when receiving a Saga message with opcode `COMMIT` for the same transaction. The `before_commit()` procedure is called before the lock-free reservation commits. The application is expected to implement this method and the applications that use lock-free reservations can choose whether to implement the `before_commit()` or `after_commit()` procedure.

`after_commit` Procedure

Syntax: `procedure after_commit(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The `after_commit` procedure is invoked after executing the Saga commit on the current participant when receiving a Saga message with opcode `COMMIT` for the same transaction. The `after_commit` procedure is called after the lock-free reservation commits. The application is expected to implement this method and the applications that use lock-free reservations can choose whether to implement the `before_commit()` or `after_commit()` procedure.

`before_rollback` Procedure

Syntax: `procedure before_rollback(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The `before_rollback` procedure is invoked before executing the Saga rollback on the current participant when receiving a Saga message with opcode `ROLLBACK` for the same transaction. The `before_rollback` procedure is called before the lock-free reservation rollbacks. The application is expected to implement this procedure and the applications that use lock-free reservations can choose whether to implement the `before_rollback` or `after_rollback` procedure.

`after_rollback` Procedure

Syntax: `procedure after_rollback(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The `after_rollback` procedure is invoked after executing the Saga rollback on the current participant when receiving a Saga message with opcode `ROLLBACK` for the same transaction. The `after_rollback` procedure is called after the lock-free reservation rollbacks. The application is expected to implement this procedure and the applications that use lock-free reservations can choose whether to implement the `before_rollback` or `after_rollback` procedure.

`forget` Procedure

Syntax: `procedure forget(saga_id IN RAW, saga_sender IN VARCHAR2, payload IN JSON DEFAULT NULL)`

Description: The Saga coordinator invokes the `forget` procedure to ask a participant to mark the Saga as forgotten. A Saga marked as forgotten cannot be finalized and requires administrative intervention. The `forget` procedure is triggered upon receiving a negative acknowledgment from the coordinator. A coordinator can respond with a negative acknowledgment to join requests. For example, if a Saga is set with a timeout and a participant join request is received after the timeout, a negative acknowledgment is sent to the participant. Another example of a negative acknowledgment is when a negative acknowledgment is sent on a join request made after a Saga has already been finalized by its initiator.

`is_join` Procedure

Syntax: `procedure is_join (saga_id IN saga_id_t, saga_sender IN VARCHAR2);`

Description: The Saga participants invoke the `is_join` procedure to disassociate themselves from the ongoing Sagas. The procedure gives the participant a choice to move ahead in this Saga or send a `REJECT` back to the initiator.

`reject` Procedure

Syntax: `procedure reject (saga_id IN saga_id_t, saga_sender IN VARCHAR2);`

Description: An initiator invokes the `reject` procedure and receives a notification if a participant rejected the invitation to join the Saga. A participant can return `FALSE` in their `is_join` method, which sends a `REJECT` message to the initiator.

`after_saga` Procedure

Syntax: `procedure after_saga(saga_id in RAW, status in varchar2)`

Description: For PL/SQL-based initiators, the `after_saga` procedure is invoked if defined in the callback package. For PL/SQL-based participants, the `after_saga` method is invoked if defined in the callback package and when event 10855 is set at level 512.

The argument `status` has the stringified values for `enum(org.eclipse.microprofile.lra.annotation.LRAStatus)`.

> **Note:**
>
> Along with the `payload`, the implemented methods supply `saga_Id` and the `saga_sender` (sender of this message) as arguments. The clients can use the `saga_Id`, `saga_sender`, and `payload` parameters for book-keeping, maintaining internal states, or both.

28.9.2 Integration with Lock-Free Reservation

Lock-free reservation enables automatic compensation (rollback) of changes to reservable columns. The reservation journal table records an entry for every update to a reservable column in the database. These entries provide the information needed for compensating

changes to the reservable columns. The reservation journal entries associated with Saga-related changes to the reservable columns are retained until the Saga is finalized.

The Saga framework uses the `saga_finalization$` dictionary table to track the set of reservable tables and their corresponding journals affected by a Saga. The creation of the reservation journal entry happens during the SQL update. For example, here is a sequence of updates and the respective `saga_finalization$` entries:

```
Update hotel_rooms set rooms = rooms - 2 where date=to_date('20201010');
Update hotel_rooms set rooms = rooms - 1 where date=to_date('20201011');
Update dinner_tables set available_tables= available_tables - 3 where
date=to_date('20201010');
```

Table 28-2 `saga_finalization$` table entries

saga_id	participant	Txn_Id	User#	reservable_table	reservation_journal	status
ABC123	Hotel	1	124	hotel_rooms	SYS_RESERV JRNL_81696	active
ABC123	Hotel	2	124	dinner_tables	SYS_RESERV JRNL_81697	active

The Saga framework performs compensating actions during a Saga rollback. Reservation journal entries provide the data that is required to take compensatory actions for Saga transactions. The Saga framework sequentially processes the `saga_finalization$` table for a Saga branch and executes the compensatory actions using the reservation journal. `Saga_finalization$` entries are marked "compensated" upon successful Saga compensation. `Saga_finalization$` entries are marked "committed" upon successful Saga commit. After a `saga_finalization$` entry is marked as "compensated" or "committed", no further actions are possible.

After a Saga is finalized (successful commit or compensation), the reservation journal entries corresponding to all reservation journals associated with the Saga are deleted. The `saga_finalization$` entries for the given Saga are also deleted.

> **See Also:**
>
> Lock-Free Reservation

28.10 AfterSaga Callbacks

An AfterSaga (or AfterLRA) callback method is called after an LRA (Long Running Action, as in a Saga) has completed. An AfterLRA method is used to notify a Saga's participants that the Saga is complete.

For a Saga in a Java-based or PL/SQL application, users can define the AfterLRA callback method. When defined, the callback method is invoked after all the participants of the Saga enter a final state; which is when all the participants involved in the Saga have successfully committed or rolled back the Saga. The Saga coordinator invokes the AfterLRA method on the initiator and all the participants, thereby triggering the `@afterLRA` annotated method for Java-based applications and the afterLRA() method for PL/SQL-based applications.

> **Note:**
>
> The initiator's AfterSaga callback is invoked by default. Participant's AfterSaga callback is only invoked when event 10855 is set at level 512.

`@afterLRA` Method

For Java-based initiators and participants, the method decorated with `@afterLRA` is triggered. The method decorated with the `@afterLRA` annotation is expected to have the following signature:

```
@AfterLRA
public void testAfterLRA(SagaMessageContext ctx, LRAStatus status) {
}
```

`SagaMessageContext` is the Saga message context object and `LRAStatus` is defined by LRAStatus (as defined by Eclipse Microprofile).

`after_saga` Method

For PL/SQL-based initiators and participants, the following method is invoked, if defined in the callback package:

```
procedure after_saga(saga_id in RAW, status in varchar2)
```

`status` has the stringified values for `enum(org.eclipse.microprofile.lra.annotation.LRAStatus)`.

The Saga coordinator attempts to invoke the AfterLRA method for all completed Sagas in an interval of 5 minutes (300 seconds). You can modify this interval using the underscore parameter: `_saga_afterlra_interval`.

Dictionary Views

Calling the user-defined AfterLRA method changes the initiator's status in the `USER_SAGAS` dictionary views to:

- `committing` or `rolling back` after the initiator submits `commit` or `rollback`.
- `committed` or `rolled back` after processing the `commit` or `rollback` for a participant and after all the participants complete `commit` or `rollback` for the initiator.

> **See Also:**
>
> USER_SAGAS in *Database Reference Guide* for views-related information