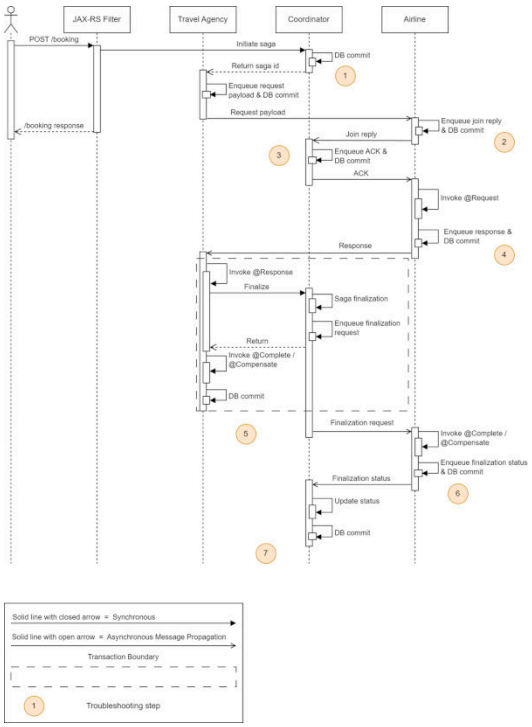
A.1 Appendix: Troubleshooting the Saga Framework

This appendix describes different ways of troubleshooting problems that may occur when you use the Saga framework with Java applications.

The Saga framework is a complex distributed system comprising multiple pluggable databases, and background and foreground database processes. The Saga framework uses database Advanced Queueing (AQ) or Transactional Event Queues (TxEventQ) as its event mesh. An event mesh seamlessly distributes events between various entities in a Saga topology. AQ message propagation and notification features enable event transmission to multiple entities in the Saga topology. The following call-return diagram depicts the lifetime of a simple Saga application that is described in the example program.

> **See Also:**
>
> Example Program

Figure 1 Saga Lifecycle and Call-return Flow



Note:

- The Saga framework uses message propagation between the initiator and the participant using a message broker as an intermediary. It also relies on AQ notification framework to allow participants and coordinator to respond to the messages received. The message broker is not depicted in this picture.
- A database commit (DB commit) differs from a Saga commit (Saga finalize), in that, a database commit commits the local database transaction at the participant or initiator databases performed for a Saga. The Saga commit, on the other hand, commits the Saga and all its transactions performed by various participants on behalf of a Saga. The Saga commit is the final commit for all local transactional changes in a Saga as a group. These transactional changes were committed by various Saga participants previously using their respective DB commits. Conversely, a Saga rollback cancels the local transactional changes for a Saga as a group by doing compensation.
- The Saga framework provides an asynchronous platform for developing Sagas. The asynchronous platform is supported by AQ messaging in the database. Certain operations, however, are synchronous and accomplished inside the database server. For example, Saga initialization is a synchronous operation. Saga finalization at the coordinator and at the initiator is another example of a synchronous operation.
- For the current version of Saga infrastructure, the initiator and coordinator are co-located in a PDB. A co-located initiator and coordinator allows the finalization of a Saga using a single database transaction encompassing both entities. See the Transaction Boundary in the diagram highlighting this aspect.

A.1.1 Tracking a Saga

This section discusses about how you can keep track of the various states in a Saga at the initiator, participant and AQ topology levels.

A.1.1.1 Saga Initiator and Participant

Saga Initiator

On the initiator's PDB, the information about ongoing Sagas is available from the `DBA_SAGAS` and `DBA_SAGA_PARTICIPANT_SET` dictionary views. The `DBA_SAGAS` view provides the current state of ongoing Sagas, and `DBA_SAGA_PARTICIPANT_SET` shows the state of individual participants in a Saga. The completed Sagas appear in the `DBA_HIST_SAGAS` view.

Saga Participant

On the participant's PDB, the information about ongoing Sagas is available from the `DBA_SAGAS` view. The completed Sagas appear in the `DBA_HIST_SAGAS` view.

A.1.1.2 AQ Topology

The following describes the AQ topology that supports the Saga framework. This information is available in the `DBA_SAGA_PARTICIPANTS` dictionary view. For the following example, let us assume that the broker named TestBroker is created on the Broker PDB.

Entity Name	Queue Name	Description
TestBroker	SAGA$_TESTBROKER_INOUT1	This queue represents the In/Out queue for a Saga broker. This queue is connected to other queues using AQ propagation.
TravelAgency	SAGA$_TRAVELAGENCY_IN_Q1	The initiator's IN queue stores incoming participant responses.
	SAGA$_TRAVELAGENCY_OUT_Q1	The initiator's OUT queue stores outgoing request messages.
TACoordinator	SAGA$_TACOORDINATOR_IN_Q1	For notification, use: "plsql://dbms_saga_adm.notify_callback_coordinator"
		The coordinator's IN queue stores incoming acknowledgements and finalization status messages.
	SAGA$_TACOORDINATOR_OUT_Q1	The coordinator's OUT queue stores outgoing acknowledgments and finalization requests.
Airline	SAGA$_AIRLINE_IN_Q1	The participant's IN queue stores the incoming payload requests and acknowledgments.
	SAGA$_AIRLINE_OUT_Q1	The participant's OUT queue stores the outbound messages bound to both initiators and coordinators.

A.1.1.3 Saga State Machine

A Saga transitions through the following states during its lifetime. For an example of this state transition, see the following section on "Troubleshooting Steps".

State	Description
Initiated	A new Saga is marked Initiated on the initiator PDB.
Joining	The Joining state in a Saga implies that the given participant is joining the Saga. This state is only relevant on the participant's PDB while the participant waits for an acknowledgment from the Saga coordinator to join the Saga.
Joined	A Saga is in the Joined state when it is successfully acknowledged by the Saga coordinator. This state is only relevant on the participant's PDB.
Finalization	A Saga undergoing commit or rollback has the Finalization state. Sagas undergoing finalization are listed in the DBA_HIST_SAGAS view.
Committed/Rolled back	A Saga that is committed or rolled back has this sate. Finalized Saga are listed in DBA_HIST_SAGAS view.
Auto Rolledback	A Saga is marked auto rolled back if it exceeds the Saga duration and is automatically rolled back by the initiator.

A.1.1.4 Saga Participant States

The DBA_SAGA_PARTICIPANT_SET view shows the state of a participant once it is associated with a Saga. The participant transitions through the following states:

State	Description
Joined	The participant has `Joined` the Saga and is processing the request payload.
Committed/Rolledback	The Saga has been finalized by the participant.
Committed/Rollback Failed	The Saga finalization has failed.
Auto Rolledback	The Saga has been automatically rolled back as it exceeded its duration.
Rejected	The participant has rejected the join Saga request.

A.1.2 Troubleshooting Steps

1. Java code enters the `@LRA` annotated method `booking()` of the `TravelAgencyController` class.

   A query on `DBA_SAGAS` on the `TravelAgency's` PDB (`Travel PDB`) shows the following state:

   Travel PDB:

   ```
   SELECT id, initiator, coordinator, status FROM dba_sagas;
   ```

id	initiator	coordinator	status
abc123	TravelAgency	TACoordinator	Initiated

   At this point, no participants have been enrolled. The Saga identifier 'abc123' has been assigned for the newly created Saga.

2. This step corresponds to the participant `Airline` formally joining the Saga by sending an ACK to the Saga coordinator. This is handled by the Saga framework and is initiated by the join message from the Saga initiator. The Saga initiator invokes the `sendRequest()` call as shown in the following code segment:

   ```
   saga.sendRequest ("Airline", bookingPayload);
   ```

   The ACK is recorded at both the coordinator and participant PDBs. The following steps: 3 and 4, represent the chronological order of operations, as shown through the Saga dictionary views on the respective PDBs.

   Airline PDB (ACK initiated)

   ```
   SELECT id, participant, initiator, status FROM dba_sagas WHERE id='abc123';
   ```

id	participant	initiator	status
abc123	Airline	TravelAgency	Joining

3. The travel coordinator (`TACoordinator`) receives the asynchronous ACK message and responds by adding the participant (`Airline`) to the participant set for the given Saga and sending a message in response.

Travel Coordinator PDB (ACK received):

```
SELECT id, participant, status FROM dba_saga_participant_set WHERE
id='abc123';
```

id	participant	status
abc123	TravelAgency	Joined

4. The `Airline` receives the ACK message from the Saga coordinator and initiates the processing of the Saga payload using its `@Request` annotated method: `handleTravelAgencyRequest()`.

   Airline PDB (ACK complete):

   ```
   SELECT id, participant, initiator, status FROM dba_sagas WHERE id='abc123';
   ```

id	participant	initiator	status
abc123	Airline	TravelAgency	Joined

   The process of join acknowledgment is conducted using AQ messaging, propagation, and notification. The join message is enqueued at the source OUT queue (`SAGA$_AIRLINE_OUT_Q1` in our example) and propagated to the Saga coordinator's IN queue (`SAGA$_TACOORDINATOR_IN_Q1`) by the way of the message broker.

5. Upon receiving a response from `Airline`, the initiator (`TravelAgency`) finalizes (commits) the Saga. The finalization process involves the following state transitions.

   Travel PDB:

   ```
   SELECT id, initiator, coordinator, status FROM dba_sagas WHERE id='abc123';
   ```

id	initiator	coordinator	status
abc123	TravelAgency	TACoordinator	Committed

6. The `Airline` receives the commit message and initiates its own commit action. This results in the Saga transitioning from `Joined` to `Committed` on the participant PDB. The `Airline` sends a message to the coordinator indicating the status of its commit.

   Airline PDB:

   ```
   SELECT id, participant, initiator, status FROM dba_sagas WHERE id='abc123';
   ```

id	participant	initiator	status
abc123	Airline	TravelAgency	Committed

7. The final step corresponds to the `TACoordinator` registering the finalization status for `Airline`. This is reflected in the `dba_saga_participant_set` view.

Travel Coordinator PDB:

```
SELECT id, participant, status FROM dba_saga_participant_set WHERE
id='abc123';
```

id	participant	status
abc123	TravelAgency	Committed

A.1.3 Diagnosing AQ Issues

As previously noted, the Saga framework uses asynchronous messaging (AQ) as its event mesh. The Saga framework relies on AQ message propagation and message notification. To diagnose issues with propagation and notification, see Managing Queues in the *Oracle Database Reference guide*.

Key elements to monitor are:

- Queue Depth for Saga queues (messages pending consumption)
- Propagation latency in the topology
- Notification latency

A.1.4 Saga Performance

The performance and throughput of Saga applications depend on the Saga framework and AQ performing at an optimal level. The database systems (PDBs) should be configured with adequate `JOB_QUEUE_PROCESSES` to facilitate AQ propagation and notification. Java applications should configure sufficient producers and consumers. See `numListeners` and `numPublishers` in the Configuration section of the Saga annotations.

In addition, the following parameters can be modified to achieve a higher level of performance.

`queue_partitions`

For higher throughput, Saga entities can deploy several queue partitions. By default, Saga entities use a single queue partition. Multiple queue partitions allow for a greater degree of parallelism for the Saga framework. All entities in the topology must have a matching number of queue partitions configured.

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

`listener_count`

By default, the Saga framework uses the AQ message notification feature for handling the message traffic on the coordinator's IN queue. Under high load, the coordinator's IN queue may become a bottleneck for performance. The `listener_count` parameter of the `DBMS_SAGA_ADM.ADD_COORDINATOR()` procedure can be revised to a number greater than 1. This enables Saga message listeners to process messages on the Saga coordinator's IN queue instead of using AQ notifications. Saga message listeners eliminate some of the

overheads of AQ message notification and provide an efficient mechanism for handling the inbound message traffic for the Saga coordinator.

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

A.1.5 Logging

The Saga framework supports the SLF4J logging facade, allowing end users to plug in logging frameworks of their own choice at deployment time. It is important, regardless of the logging framework, to ensure the proper permissions and best practices are applied to the log files.