

Oracle®

MicroTx - Distributed Transactions Developer's Guide



Release 26.1

G36530-01

May 2026

ORACLE®

Oracle MicroTx - Distributed Transactions Developer's Guide, Release 26.1

G36530-01

Copyright © 2022, 2026, Oracle and/or its affiliates.

Primary Author: Sylaja Kannan

Contributing Authors: Tulika Das

Contributors: Todd Little, Deepak Goel, Brijesh Kumar Deo, Bharath MC, Pruthvithej R, Satyanarayana Chillale, Atul Dhiman, Tushar Shaily, Chandrashekar Venkatachar, Deepak Kesawani, Himanshu Gaur, Shivanshu Singh

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

1 About MicroTx Distributed Transactions

1.1	How MicroTx Distributed Transactions Works	2
1.2	Components of MicroTx Distributed Transactions	3
1.3	About the Distributed Transaction Protocols	4
1.3.1	XA Transaction Protocol	5
1.3.2	Saga Transaction Protocol	6
1.3.3	Try-Confirm/Cancel Transaction Protocol	8
1.4	Limits	10
1.5	About Choosing a Transaction Protocol	10

2 Install

2.1	Workflow to Install MicroTx	2
2.2	Access MicroTx Distributed Transactions	3
2.3	Find IP Address of Istio Ingress Gateway	4
2.4	About MicroTx Distributed Transactions Client Libraries	4

3 Develop Applications with XA

3.1	About MicroTx Distributed Transactions Library for XA	2
3.2	Workflow to Develop Applications with XA	2
3.3	Plan Your Resource Manager	3
3.3.1	Supported Resource Managers	3
3.3.2	Supported Drivers for Resource Managers	4
3.3.3	Optimizations for a Non-XA Resource	5
3.3.4	Common Resource Manager for Multiple Apps	6
3.3.5	Configure Multiple Resource Managers for a Single App	7
3.3.6	About Dynamic Recovery for XA Transactions	7
3.4	Provide the Resource Manager Connection Details	8
3.5	Configure PostgreSQL as Resource Manager	9
3.6	Manage XA Transactions	10
3.6.1	Set Transaction Timeout	11
3.6.2	About Global and Local Transactions	11
3.6.2.1	About MicroTx Distributed Transactions Client Library Annotations	12

3.6.2.2	About Transaction Promotion	12
3.6.2.3	Enable Transaction Promotion	13
3.6.3	About Asynchronous Calls to Participant Apps	13
3.6.4	Subscribe to Receive XA Transaction Notifications	15
3.7	Required User Privileges	18
3.8	Configure Library Properties for JAX-RS and Node.js Apps	18
3.9	About @Transactional	20
3.10	Develop Spring Boot Apps with XA	23
3.10.1	Configure Library Properties for Spring Boot Apps	23
3.10.2	Configure Spring Boot App as Transaction Initiator	25
3.10.3	Configure Spring Boot App as Transaction Participant	31
3.10.3.1	Configure Spring Boot Apps with an XA-Compliant Resource Manager	32
3.10.3.2	Configure Spring Boot App with Multiple XA-Compliant Resource Managers	36
3.10.3.3	Configure Spring Boot App with a Non-XA JDBC Resource	41
3.10.3.4	Configure Spring Boot App with a Non-XA and Non-JDBC Resource	46
3.10.3.5	Configure JPA-Based Spring Boot Apps with XA-Compliant Resource Manager	50
3.11	Develop JAX-RS Apps with XA	59
3.11.1	About MicroTx library for JAX-RS Apps	60
3.11.2	Configure JAX-RS App as Transaction Initiator	60
3.11.3	Configure JAX-RS App as Transaction Participant	62
3.11.3.1	Configure JAX-RS App with an XA-Compliant Resource Manager	63
3.11.3.2	Configure JAX-RS App with Multiple XA-Compliant Resource Managers	66
3.11.3.3	Configure JAX-RS App with a Non-XA JDBC Resource	70
3.11.3.4	Configure JAX-RS App with a Non-XA and Non-JDBC Resource	74
3.11.4	Configure JPA-Based Java App as Transaction Participant	76
3.11.4.1	Configure JPA-Based Java App with an XA-Compliant Resource Manager	76
3.12	Configure Java Apps to Leverage Transactional Event Queues	86
3.12.1	Required Privileges	87
3.12.2	Configure JAX-RS Participant App to Leverage Transactional Event Queues	87
3.12.3	Configure Spring REST Participant App to Leverage Transactional Event Queues	90
3.13	Develop Node.js Apps with XA	93
3.13.1	Configure Node.js App as Transaction Initiator	94
3.13.2	Configure Node.js App as Transaction Participant	96
3.13.2.1	Configure Node.js Apps with an XA-Compliant Resource Manager	96
3.13.2.2	Configure Node.js Apps with a Non-XA Resource	99
3.14	Develop ORDS App with XA	100
3.14.1	Prerequisites	101
3.14.2	Run MicroTx Distributed Transactions PL/SQL Library	102
3.14.3	Configure PL/SQL Library Properties for ORDS Apps	102

3.14.4	Privileges	103
3.14.5	Configure ORDS App as Transaction Initiator	104
3.14.6	Configure ORDS App as Transaction Participant	106
3.14.7	MicroTx Distributed Transactions PL/SQL Library Functions	110
3.14.7.1	TMM_BEGIN_TX Function	110
3.14.7.2	TMM_GET_TX_STATUS Function	111
3.14.7.3	TMM_COMMIT_TX Function	112
3.14.7.4	TMM_ROLLBACK_TX Function	113
3.14.7.5	TmmStart Function	114
3.14.7.6	TmmEnd Function	116
3.15	Include Oracle Blockchain Platform in XA Transactions	116

4 Develop Applications with Saga

4.1	Workflow to Develop Applications with Saga	1
4.2	Propagate Oracle-Tmm-Tx-Token for multiple REST API calls	2
4.3	Develop Java Apps with Saga	3
4.3.1	Develop Helidon Applications with Saga	4
4.3.2	Configure MicroTx Distributed Transactions Library Properties for Java Apps	4
4.3.3	Develop Spring Boot Applications with Saga	6
4.3.4	Develop Micronaut Apps with Saga	8
4.3.5	Develop Java Apps that Use Saga and Lock-Free Reservation	9
4.4	Develop Node.js Apps with Saga	11
4.4.1	Configure Library Properties for Node.js Apps	12
4.4.2	Configure Node.js Apps as Transaction Initiator or Participant	13

5 Develop Applications with TCC

5.1	Guidelines	1
5.2	Workflow to Develop Applications with TCC	2
5.3	Configure Library Properties	2
5.4	About Transaction Timeout	3
5.5	Develop Spring Boot Apps with TCC	4
5.5.1	Configure Spring REST App as Transaction Initiator	4
5.5.2	Configure Spring Boot App as Transaction Participant	7
5.6	Develop JAX-RS Apps with TCC	9
5.6.1	Configure JAX-RS App as Transaction Initiator	10
5.6.2	Configure JAX-RS App as Transaction Participant	11
5.7	Develop Node.js Apps with TCC	13
5.7.1	Configure Node.js App as Transaction Initiator	13
5.7.2	Configure Node.js App as Transaction Participant	15
5.8	Develop Python Apps with TCC	17

5.8.1	Configure Python App as Transaction Initiator	17
5.8.2	Configure Python App as Transaction Participant	19

6 Develop Tuxedo Apps with XA

6.1	Run Tuxedo App on Linux Host	1
6.1.1	Prepare the Environment	1
6.1.2	Install Patches	2
6.1.3	Verify the Set Up	4
6.2	Run Tuxedo App in Kubernetes Cluster	6
6.2.1	Start Tuxedo Sample App in a Docker Container	6
6.2.2	Update the YAML Files for Tuxedo App	6

7 Manage the Transaction Coordinator

7.1	About Session Affinity	1
7.1.1	Enable Session Affinity	2
7.2	Enable Caching	3
7.3	About Transaction Recovery	4
7.4	General Syntax of Commands	4
7.5	Scale up or down	5
7.6	Update	6
7.7	Uninstall	7

8 Manage Transactions Using the Web Console

8.1	Access the Web Console	1
8.2	View Details of a Transaction	2
8.3	View the Total Number of Transactions	3
8.4	View the Number of Recent Transactions	4
8.5	View Coordinator Health	4
8.6	View Console Health	5
8.7	View Configuration Properties	5
8.8	Manage XA Transactions	5
8.8.1	Commit an XA Transaction	6
8.8.2	Roll Back an XA Transaction	6
8.8.3	Delete an XA Transaction	6
8.9	Manage TCC Transactions	6
8.9.1	Confirm a TCC Transaction	7
8.9.2	Cancel a TCC Transaction	7
8.10	Manage Saga Transactions	7
8.10.1	Complete a Saga Transaction	8

9 Deploy Sample Applications

9.1	About XA Sample Application	2
9.1.1	Scenario: Withdraw and Deposit an Amount	3
9.2	About the Sample Saga Application	4
9.3	About the Sample TCC Application	6

10 Monitor and Trace Transactions

10.1	Trace	1
10.1.1	Install Jaeger	2
10.1.2	Perform Distributed Tracing with Jaeger	3
10.1.3	Install Kiali	4
10.1.4	List of Trace Headers	5
10.2	Monitor Performance	6
10.2.1	Set Up Prometheus and Grafana	7
10.2.2	View Metrics with Prometheus and Grafana	8
10.3	Install and Configure Alertmanager	9
10.4	Logs	10
10.4.1	Define the Syntax of Log Messages	11
10.4.2	Sample Log Messages	15
10.4.3	Enable Logs for MicroTx Distributed Transactions Node.js Library	17

11 Troubleshooting

11.1	Cannot retrieve transaction details from the MicroTx Distributed Transactions coordinator	1
11.2	Unable to retrieve details about the MicroTx Distributed Transactions coordinator and console health	2
11.3	No data is displayed in the Console	2
11.4	Handle a spike in traffic for Helidon apps	3

A Deploy Your Application

A.1	Build the Docker Image	A-1
A.2	Push App Image to a Remote Repo	A-2
A.3	Create Helm Files	A-3
A.4	Install Your Application	A-3

About MicroTx Distributed Transactions

Oracle Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions) enables enterprise users to adopt and increase use of microservices architecture for mission-critical applications by providing capabilities that make it easier to develop, deploy, and maintain data consistency in such applications.

Although microservice architecture provides many benefits, it is difficult to ensure data consistency for requests that span multiple services. Currently, service developers can include compensating transactions in their application code or use Saga for eventual consistency. However, these solutions are error prone and require advanced coding skills. It is also difficult to troubleshoot and manage transactions that span polyglot microservices. The complexity increases further when each microservice uses an individual database to manage their data.

As organizations rush to adopt microservices architecture, they often run into problems associated with data consistency as each microservice typically has its own database. In monolithic applications, local transactions were enough as there were no other sources of data that needed to be consistent with the database. An application would start a local transaction, perform some updates, and then commit the local transaction to ensure the application moved from one consistent state to another. Once the application's state is spread across multiple sources of data, some factors need to be considered. What happens if updates succeed in one microservice, but it fails in another microservice as part of the same request? One solution is to use a distributed transaction that spans the sources of data used by the microservices involved in a request. Oracle Transaction Manager for Microservices provides a transaction coordination microservice and libraries to maintain consistency in the state of microservices participating in a transaction.

MicroTx Distributed Transactions ensures consistency of transactions across distributed microservices applications. It performs the following actions:

- Manages transactions and provides consistency across polyglot microservices.
- Supports several distributed transaction protocols, such as XA, Saga based on Eclipse MicroProfile Long Running Actions (LRA) and Try-Confirm/Cancel (TCC). Based on your business requirements and the level of consistency that's required, you can select a suitable transaction protocol for your application.
- Addresses critical needs for enterprise customers to provide a highly-available, scalable, and secure solution.
- Integrates with powerful cutting-edge technologies, such as Jaeger, Kiali, Prometheus, and Grafana. It provides you with a variety of options for activities, such as data visualization, data monitoring, transaction tracing, which enables advanced and efficient troubleshooting and data management operations.
- Runs in a Kubernetes cluster or Docker Swarm environments along with other microservices in on-premises, cloud, and hybrid environments.
- Works with popular programming languages and application frameworks, such as Node.js and Java.
- Supports inclusion of Oracle Tuxedo services that are written in C, C++, and COBOL languages.

- Supports inclusion of Oracle Database resident services, written in PL/SQL, in a global XA transaction with other microservices.

Topics

- [How MicroTx Distributed Transactions Works](#)
To use MicroTx Distributed Transactions, install MicroTx Distributed Transactions and then integrate the MicroTx Distributed Transactions client libraries with your application code to manage transactions.
- [Components of MicroTx Distributed Transactions](#)
MicroTx Distributed Transactions contains two components: the transaction coordinator and the MicroTx Distributed Transactions library.
- [About the Distributed Transaction Protocols](#)
MicroTx Distributed Transactions supports the following distributed transaction protocols:
- [Limits](#)
The following limits apply only to Transaction Manager for Microservices Free release and not to Transaction Manager for Microservices Enterprise Edition.
- [About Choosing a Transaction Protocol](#)
Use the information provided to select a transaction protocol for your application based on your business requirements.

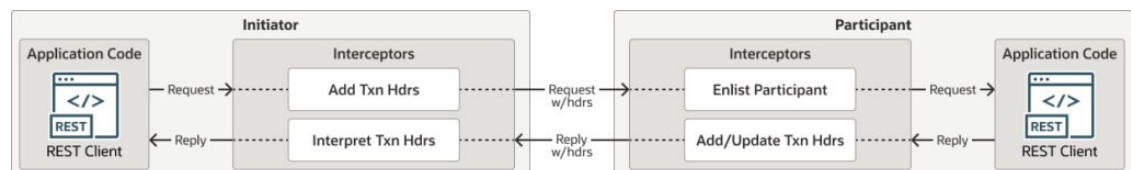
1.1 How MicroTx Distributed Transactions Works

To use MicroTx Distributed Transactions, install MicroTx Distributed Transactions and then integrate the MicroTx Distributed Transactions client libraries with your application code to manage transactions.

About interceptors provided by the MicroTx Distributed Transactions client libraries

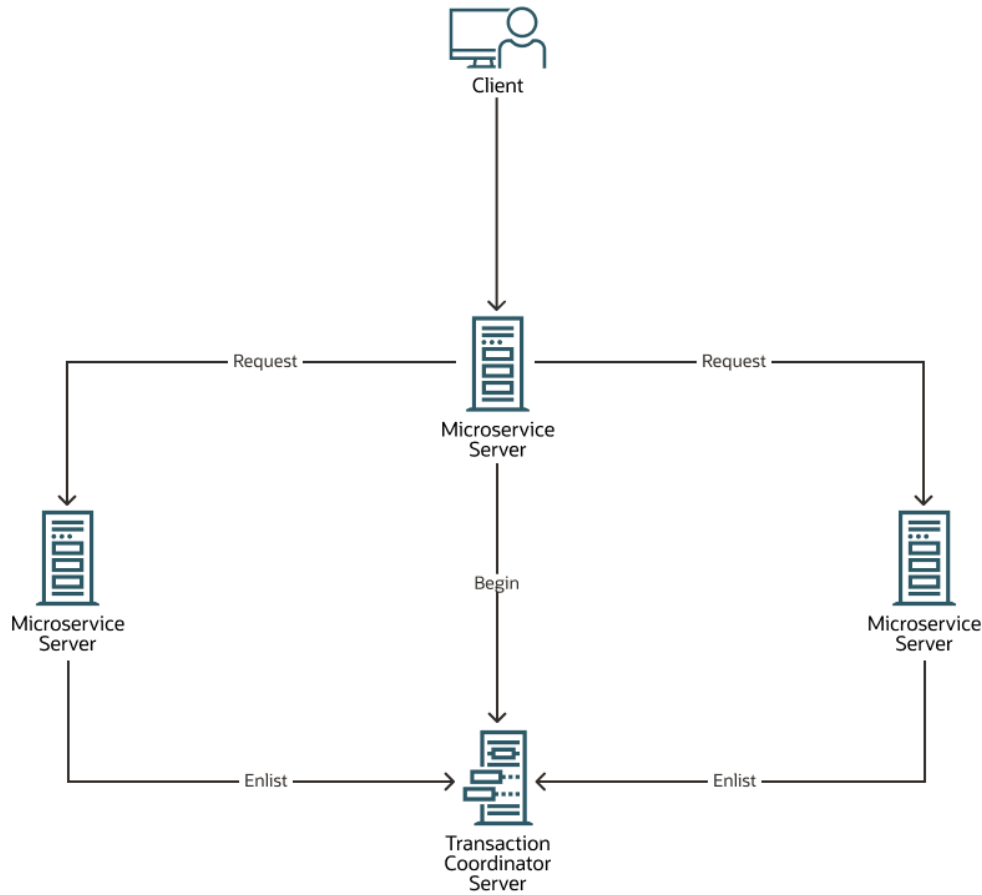
The MicroTx Distributed Transactions client libraries provide interceptors to intercept both incoming and outgoing REST calls, as well as their requests and responses. These interceptors use headers to propagate the transaction context which enable the participant microservices to automatically enlist in a transaction. The interceptors also ensure that the appropriate transaction headers are propagated in any outgoing REST call.

The following image shows the typical flow of request and responses and the role of the interceptor provided by the MicroTx Distributed Transactions libraries.



When a microservice, that uses the MicroTx Distributed Transactions client libraries, makes an outbound REST request, the library's interceptors add transaction headers to the outbound request if the microservice has started a distributed transaction or is currently participating in a distributed transaction. When a microservice receives a request, the interceptors in the recipient service identify the transaction headers and automatically enlist as a participant in the distributed transaction.

Here's a typical transaction workflow when you use MicroTx Distributed Transactions. The following figure shows how MicroTx Distributed Transactions communicates with your application microservices to handle transactions.

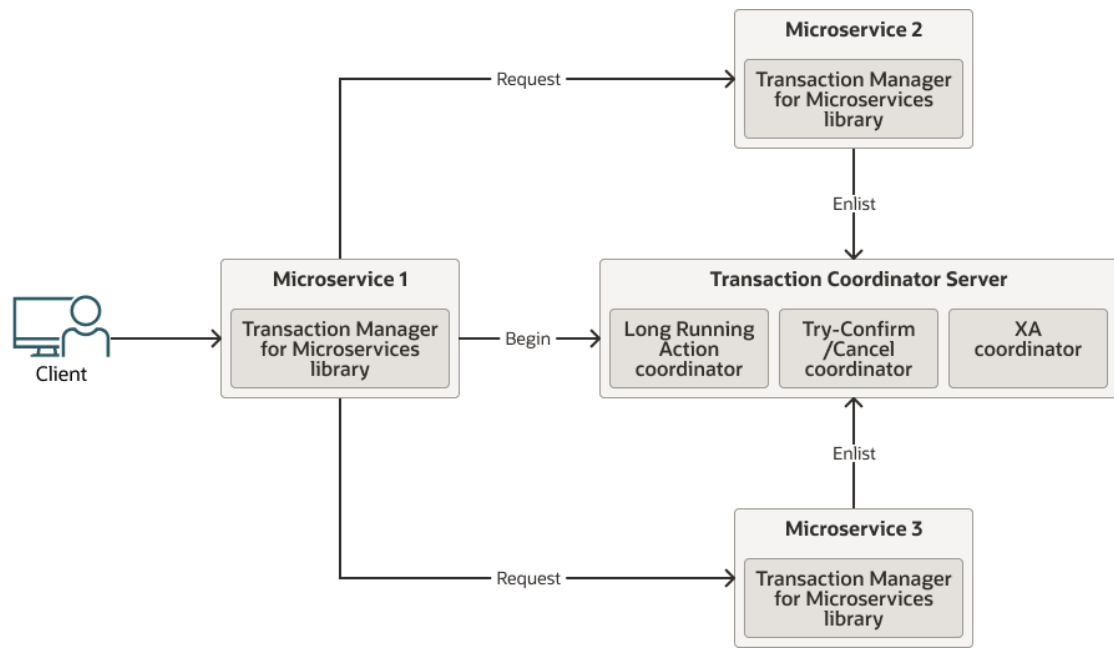


1. Application developers use functions present in the MicroTx Distributed Transactions library with their application code.
2. When a microservice or client initiates a transaction, it calls functions in the MicroTx Distributed Transactions library to start a distributed transaction.
3. MicroTx Distributed Transactions library includes headers that enable the participant services to automatically enlist in the transaction.
4. After all the tasks associated with the original request made by the initiator service are complete, the initiator service requests the transaction coordinator to either commit or roll back all the changes.
5. The transaction coordinator sends a call to each participant service to either commit or roll back the changes made by the participants as part of the distributed transaction.

1.2 Components of MicroTx Distributed Transactions

MicroTx Distributed Transactions contains two components: the transaction coordinator and the MicroTx Distributed Transactions library.

MicroTx Distributed Transactions, a containerized microservice, runs along with your application microservices. The following figure shows how the components of MicroTx Distributed Transactions interact with your application microservices.



Transaction Coordinator Server

The transaction coordinator manages transactions amongst the participant services.

MicroTx Distributed Transactions supports internal memory, Oracle Database, and etcd as a data store for persistence of transaction state.

MicroTx Distributed Transactions library

Application microservices provide the business logic and demarcate transaction boundaries. These services participate in a distributed transaction. They use MicroTx Distributed Transactions APIs to manage their distributed transactions.

Application developers use different parts of the MicroTx Distributed Transactions client library depending on the following factors:

- The development framework of the microservice, such as Helidon or Node.js.
- The selected transaction protocol, such as XA, Saga, or TCC.
- Whether the application initiates a transaction or participates in the transaction.
 - Transaction initiator service - These applications start and end a transaction. In the preceding figure, Microservice 1 is the transaction initiator service and it sends a request to MicroTx Distributed Transactions to begin the transaction.
 - Transaction participant service - These applications only join the transaction. They do not initiate the transaction. In the preceding figure, Microservice 2 and Microservice 3 are the transaction participant services that are involved in the transaction.

1.3 About the Distributed Transaction Protocols

MicroTx Distributed Transactions supports the following distributed transaction protocols:

- XA protocol, which is based upon The Open Group's XA specification. For details about the specification, see <https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf>.

- Saga protocol, which is based on the Eclipse MicroProfile LRA specification. For details about the specification, see <https://download.eclipse.org/microprofile/microprofile-lra-1.0-M1/microprofile-lra-spec.html>.
- Try-Confirm/Cancel (TCC) protocol

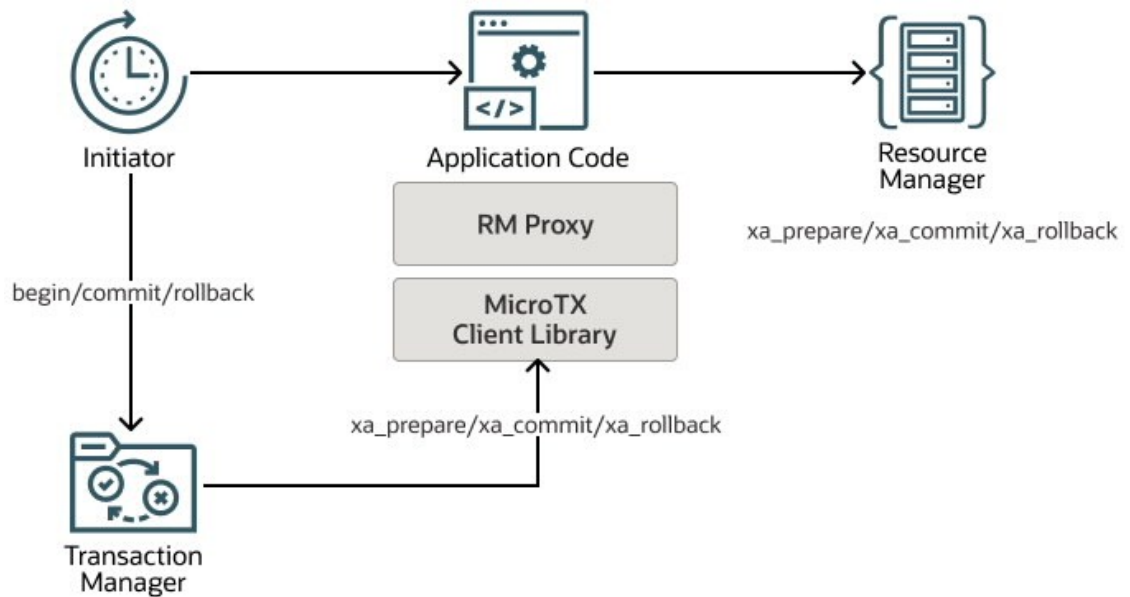
Use XA when strong consistency is required, similar to consistency provided by the local database transactions, where all the ACID properties of a transaction are present. For example, financial applications. Use the Saga protocol for transactions that may take a long time to complete. You can use the Saga protocol to mitigate locking issues. The TCC protocol fits well for applications that use a reservation model, such as airline seats or hotel rooms. Both Saga and TCC support long running transactions. Saga is far more general, but requires application specific actions for both completing a successful Saga and compensating a failed Saga. Whereas, compensation in TCC is performed by deleting the reservation, and then returning whatever was reserved to the pool of available resources.

- [XA Transaction Protocol](#)
An application using XA, must demarcate the transactions boundaries. MicroTx Distributed Transactions commits or rolls back the transaction.
- [Saga Transaction Protocol](#)
- [Try-Confirm/Cancel Transaction Protocol](#)
The Try-Confirm/Cancel (TCC) transaction protocol holds some resources in a reserved state until the transaction is either confirmed or canceled. If the transaction is canceled, the reserved resources are released and are available in the inventory.

1.3.1 XA Transaction Protocol

An application using XA, must demarcate the transactions boundaries. MicroTx Distributed Transactions commits or rolls back the transaction.

In the XA protocol, participant microservices must use the MicroTx Distributed Transactions client libraries which registers callbacks and provides implementation of the callbacks for the resource manager. As shown in the following image, MicroTx Distributed Transactions communicates with the resource managers to commit or roll back the transaction. MicroTx Distributed Transactions connects with each resource manager involved in the transaction to prepare, commit, or rollback the transaction. The participant service provides the credentials to the coordinator to access the resource manager. As shown in the following figure, MicroTx Distributed Transactions client libraries provide a resource manager proxy (RM proxy). The proxy eliminates the need for the coordinator to have resource manager specific libraries, which would be the normal case in XA. When the transaction coordinator needs to prepare, commit, or rollback the transaction for a participant's resource manager, it makes a callback to the microservice and the proxy relays the request to the resource manager being used by the microservice. These REST-based callbacks allow the transaction coordinator to be agnostic to the resource manager used by the microservice.

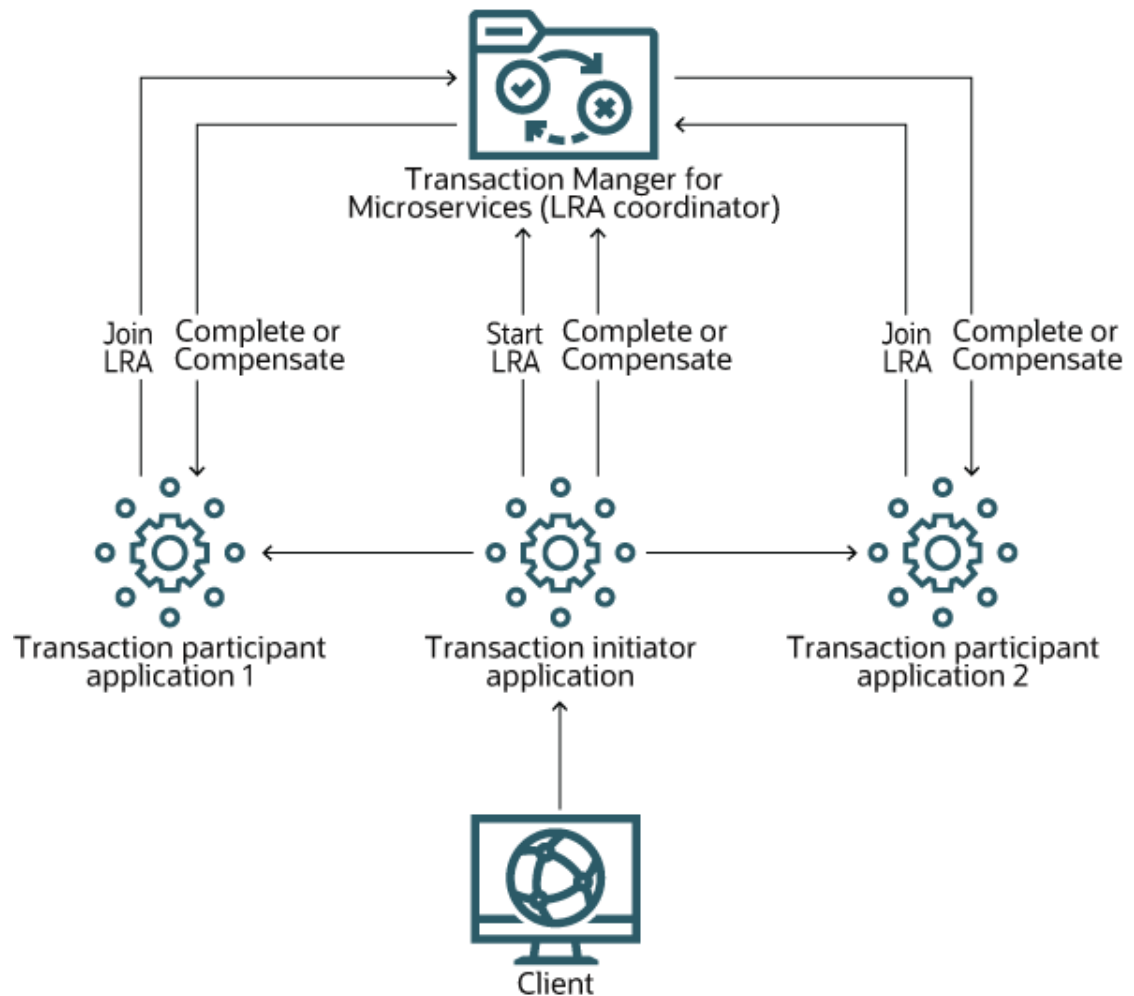


1. Initiator starts the distributed transaction
2. Called microservices enlist in the transaction
3. Initiator asks transaction manager to commit or rollback the transaction
4. If the initiator decided to commit, the transaction manager asks each microservice to prepare
 - a. If all participants successfully prepare, they are all asked to commit
 - b. If any of the participants fail to prepare, they are all asked to rollback
5. If the initiator decided to rollback the transaction, the transaction manager asks each microservice to rollback

To understand how the communication takes place between the microservices, MicroTx Distributed Transactions client libraries, and the coordinator, see [About XA Sample Application](#).

1.3.2 Saga Transaction Protocol

The following image describes how the microservices communicate with each other and with MicroTx Distributed Transactions when you use the Saga transaction protocol, which is based on Eclipse MicroProfile Long Running Actions (LRA).



Let's understand how the microservices communicate with each other to process a sample transaction.

1. The transaction initiator service calls the MicroTx Distributed Transactions Saga coordinator and passes its callback URIs to begin and enlist in the Saga transaction.
2. The transaction initiator service calls one or more participant services by passing the ID of the Saga in headers.
3. The other participant services call MicroTx Distributed Transactions and enlist or join the Saga transaction. When participants join the Saga, they provide callback URIs including ones for completing and compensating their part of the Saga.
4. The transaction initiator service calls MicroTx Distributed Transactions to either complete or compensate the transaction.
5. MicroTx Distributed Transactions calls each participant service's complete callback URI or compensate callback URI depending upon whether the transaction initiator service asks to complete or compensate the transaction.

Each participant uses local transactions that are independent from each other. Since the Saga transaction protocol uses local transactions, there are periods when the overall state of the system is inconsistent while the goal of the transaction is to achieve consistency at the end of the transaction. This is because the local transactions complete or compensate independently. As a result, there are periods when one or more local transactions are completed or compensated while others have not. Because of the lack of locking and isolation, other

systems or users will be able to see these inconsistent states and potentially make faulty decisions based upon those inconsistent states.

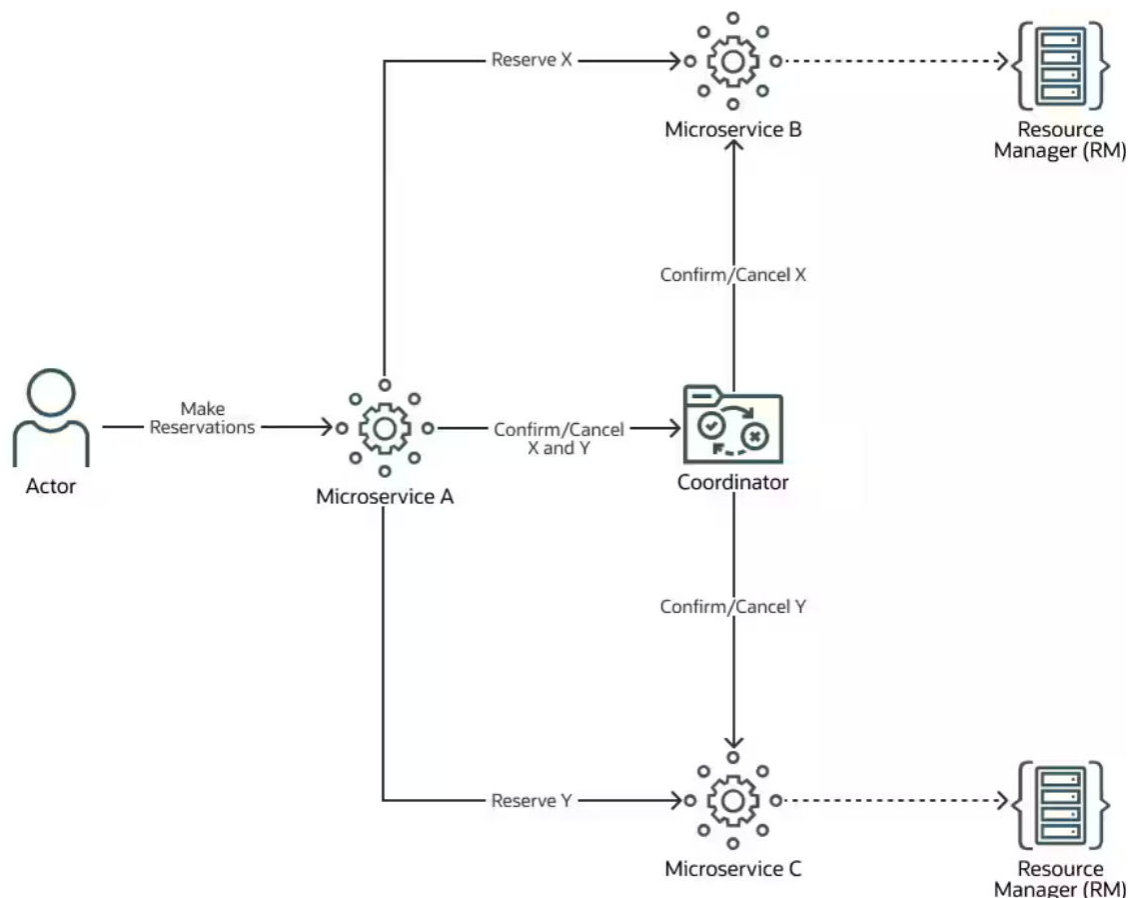
1.3.3 Try-Confirm/Cancel Transaction Protocol

The Try-Confirm/Cancel (TCC) transaction protocol holds some resources in a reserved state until the transaction is either confirmed or canceled. If the transaction is canceled, the reserved resources are released and are available in the inventory.

The TCC transaction protocol relies on the basic HTTP verbs: POST, PUT, and DELETE. Ensure that your application conforms to the following guidelines:

- The transaction initiator service must use the POST HTTP method to create a new reservation. As a response to this request, the transaction participant services must return a URI representing the reservation. The MicroTx Distributed Transactions client libraries places the URI in MicroTx Distributed Transactions specific headers to ensure that the URI is propagated up the call stack.
- This protocol relies upon the participant services to ensure that all participant services either confirm their reservations or cancel their reservations. The URIs must respond to the PUT HTTP method to confirm a reservation, and to the DELETE HTTP method to cancel a reservation.

The following image describes how microservices and MicroTx Distributed Transactions interact with each other in a TCC transaction.



Microservice A is a transaction initiator service. It starts and ends a transaction. It sends a request to participant services which indicates that the participant service should be part of the transaction.

Microservice B and C are the participant services. These services only join an existing transaction. They do not initiate a transaction.

Try Phase

In the TCC protocol, a transaction initiator services asks other participant microservices to reserve resources. During the try phase, MicroTx Distributed Transactions library collects all the accepted reservations. This includes reservations made by the participant services. By the time the initiator (in the example image above, Microservice A) completes making reservations with Microservice B and Microservice C, the MicroTx Distributed Transactions library collects all the reservations. At this point the initiator can decide to confirm the reservations, cancel the reservations, or ignore the reservations which would let timeouts eventually cancel the reservations.

Confirm/Cancel Phase

Based on the business logic provided in the initiator service, it can decide to either confirm all the reservations or cancel all the reservations. When the initiator and all participants have acquired the required reservations, the initiator service sends a request to MicroTx Distributed Transactions to confirm all the reservations. Based on its business logic, if the initiator service decides that it does not want or cannot use the reservations made, it requests the MicroTx Distributed Transactions to cancel all the reservations. What constitutes a reservation is completely up to the application.

Let us look at a simple microservice that allows reserving and purchasing a seat for a performance. Seats would have a state which could either be AVAILABLE, RESERVED, or SOLD. The try phase would have changed the state of the seat to RESERVED from AVAILABLE. The confirm phase would change the state from RESERVED to SOLD, assuming that payment was made successfully. The cancel phase would change the state from RESERVED to AVAILABLE. To prevent failure of the confirm step when a payment has not been completed successfully, during the Try phase, the microservice should obtain payment authorization to ensure the payment can be made.

Let us consider another example where an application reserves a certain quantity, such as items in an inventory or funds from an account. In this case, during the Try phase the application might deduct the reserved quantity from the available quantity and add a record of the reservation to the database. During the confirm phase, the reservation record is deleted. During the cancel phase, the amount in the reservation record is added back to the total inventory and the reservation record is deleted.

The following steps describe the successful path of a TCC transaction among microservices and MicroTx Distributed Transactions. In case of failures, the initiator service calls cancel instead of confirm.

1. The transaction initiator service, Microservice A, makes a MicroTx Distributed Transactions client library call to begin the TCC transaction.
2. The transaction initiator service invokes POST on Microservice B, a participant service, to reserve a resource X.
3. Microservice B reserves the required resources, and then returns a URI representing its reservation to Microservice A, the transaction initiator.
4. The transaction initiator service invokes POST on Microservice C, a participant service, to reserve a resource Y.
5. Microservice C reserves the required resources, and then returns a URI representing its reservation to Microservice A, the transaction initiator.

6. Microservice A, the transaction initiator service, calls MicroTx Distributed Transactions to either confirm or cancel the reservations.
7. MicroTx Distributed Transactions calls PUT to confirm or DELETE to cancel on all the URIs (reservations) to complete the transaction.
8. The participant services confirm or cancel the resources, and then return the HTTP response code 200 to MicroTx Distributed Transactions.
9. MicroTx Distributed Transactions returns a successful status to the transaction initiator, Microservice A. If MicroTx Distributed Transactions does not receive 200 status from one or more participants, then it returns an error message.

1.4 Limits

The following limits apply only to Transaction Manager for Microservices Free release and not to Transaction Manager for Microservices Enterprise Edition.

- **Transaction Throughput:** The Free edition permits 4800 transactions per hour across all the transaction protocols. If you exceed this limit, the HTTP 429: Too Many requests error is displayed. The time period is considered from the moment you start MicroTx Distributed Transactions.
- **Data Store:** The Free edition permits only in-memory storage type to store transaction data . Use it for local development, testing, and samples. It is not intended for production because data is lost when the coordinator restarts.

1.5 About Choosing a Transaction Protocol

Use the information provided to select a transaction protocol for your application based on your business requirements.

Different business use cases require different levels of consistency. For example, financial applications that move funds require strong global consistency. The XA transaction protocol is a good fit for such applications as XA offers the best transaction consistency with the least amount of developer effort. On the other hand making travel reservations typically doesn't require this level of consistency, so Saga may be a better fit. Saga transactions provide the most flexibility at the cost of developer complexity.

The following table lists a few parameters to help you choose a transaction protocol for your application.

Parameters	XA	Saga	TCC
Transaction consistency level	Strongest	Eventual	Strong
Dirty reads	No	Yes	No
App development complexity	Low	High	Medium
Auto rollback on timeouts or errors	Yes	Yes	Yes
Transaction performance	Good	Better	Best
Locks held during the transaction	Yes	No	No

XA participants hold locks for the duration of the transaction. Saga and TCC use local transactions that only span the duration of the participant's business logic.

The XA Transaction Protocol

Use this protocol for applications when the programming model places minimal requirements on the application, with the application only determining the boundaries and outcome of a transaction. You can also use XA when the service must meet ACID requirements, which requires all participants to move from one consistent state to another, with complete isolation and serializability.

To ensure serializability, resource managers lock the resources that have been read, written, or deleted while the transaction is in process. This means that other transactions using those same resources must wait until those locks are released. This serialization of requests waiting for these locks can significantly limit the performance of an application.

Another potential performance issue with XA is the additional latency it adds to a transaction. The impact depends upon the latency of the actual business request and the latency of the XA operations. For example, if a business request spanning several microservices takes 800 milliseconds and the XA operations add another 200 milliseconds, it may not have a major impact on the importance. However, if a business request takes 50 milliseconds, but the latency of XA operations adds an additional 200 milliseconds, that would have a significant impact on the application's performance.

The Saga Transaction Protocol

Use this protocol for applications where it might not be feasible or appropriate to use XA transaction protocol. As XA transactions involve locks on resources, it is recommended that XA transactions are relatively short lived involving only machine-to-machine interactions. Saga protocol is a better fit when users are involved in the decision making process for a transaction or for long workflows that may execute over minutes to hours or more.

Since Saga protocol does not lock resources, they offer a major advantage as they do not introduce serialization performance issues. Avoiding serialization issues is great for performance, however Saga places some significant burdens on the application. When an Saga transaction is aborted or canceled, the application developer must provide the code to perform the appropriate compensating action. This may sound easy as one can trivially compensate a deposit with a withdrawal. Yet if another intervening withdrawal has taken place, it is conceivable that there aren't enough funds to make the compensating withdrawal. In this case it is likely that the compensating action would fail leaving the transaction with a heuristic outcome. Many other cases exist where it may be extremely difficult or impractical to implement compensating actions. It is the responsibility of the application developer to create the compensating actions, and it may be difficult to test the compensating actions under all failure scenarios.

To use the advantages offered by both Saga and XA transaction protocols, you can nest an XA transaction within a Saga transaction. Let's consider an application which books movie tickets. The microservices that reserve the seats, use the Saga transaction protocol. The microservices that make the payment for the reserved seats, use XA transaction. In this way you can utilize the advantages offered by both Saga and XA transaction protocols and improve performance.

Try-Confirm/Cancel (TCC) Transaction Protocol

Use this protocol when application business model supports reservations. For example, a travel agency application which books a flight, rental car, hotel.

The TCC transaction protocol guarantees the same global consistency that the XA transaction protocol provides, yet with limits on the type of application that can leverage the TCC transaction protocol. TCC works only with application resources that can be held in reserve. For example, flight or hotel reservations. With each reservation, the system moves from one

consistent state to another. The protocol is completely scalable as there are no imposed serialization constraints. Similar to XA, TCC is easy for the developer to utilize as the developer only needs to demarcate the transaction boundaries and determine the outcome of the transaction. The transaction coordinator handles the workflow to ensure all participant services either confirm or cancel the transaction, which further minimizes the responsibility placed on the application code.

2

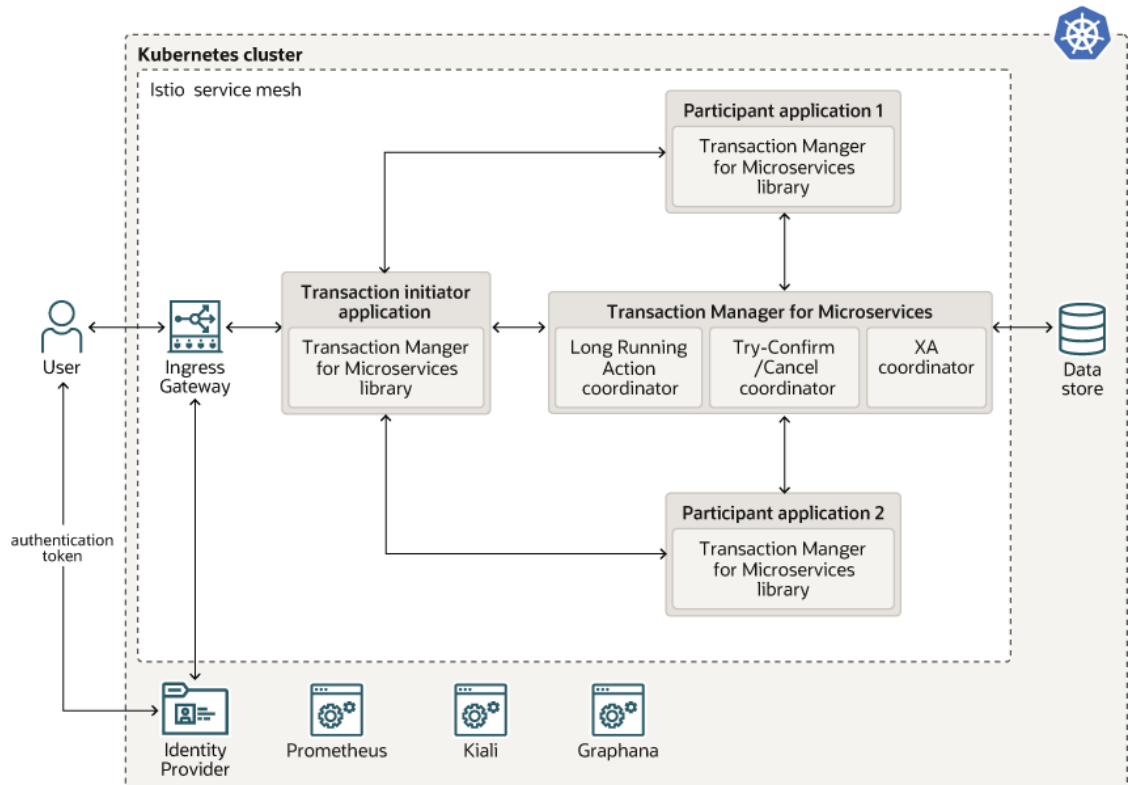
Install

In Kubernetes, you can install MicroTx Distributed Transactions within a service mesh or without it.

The installation bundle provides example Helm charts with sample values which you can use as a reference to install MicroTx Distributed Transactions. This section provides instructions to install MicroTx Distributed Transactions on a Kubernetes cluster with and without using the Istio service mesh.

You can create a similar configuration to install MicroTx Distributed Transactions in other supported environments. If you are using another service mesh in a Kubernetes cluster, create your own Helm charts.

The following image shows a sample deployment where MicroTx Distributed Transactions is installed in a Kubernetes cluster within an Istio service mesh along with other microservices.



Istio is a service mesh that provides a separate infrastructure layer to handle inter-service communication. Network communication is abstracted from the services themselves and is handled by proxies. Istio uses a sidecar design, which means that the communication proxies run in their own containers beside every service container. Envoy is the proxy that is deployed as a sidecar inside the microservices container. All communication inside the service mesh is done through the Envoy proxies.

See [Workflow to Install MicroTx](#).

- [Workflow to Install MicroTx](#)
Use the following workflow as a guide to install, configure, and use Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions).
- [Access MicroTx Distributed Transactions](#)
To access MicroTx Distributed Transactions, specify the port number, host name, and protocol that you want to use to access. Oracle recommends that you use HTTP protocol only in test or development environments. In production environments, you must use HTTPS protocol.
- [Find IP Address of Istio Ingress Gateway](#)
Before you start a transaction, you must note down the external IP address of the Istio ingress gateway.
- [About MicroTx Distributed Transactions Client Libraries](#)
After installing MicroTx Distributed Transactions, you can integrate MicroTx Distributed Transactions client libraries with your application code or run sample applications to understand how MicroTx Distributed Transactions works.

2.1 Workflow to Install MicroTx

Use the following workflow as a guide to install, configure, and use Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions).

Task	Description	See
Understand the requirements and select a transaction protocol for your application	Plan the installation and setup of MicroTx Distributed Transactions based on your business requirements.	Plan
Download the installation bundle	The installation bundle contains the MicroTx Distributed Transactions image and other required files.	Download the Installation Bundle
Complete the authentication and authorization requirements	Set up an identity provider and create an access token.	About Authentication and Authorization
Push the MicroTx Distributed Transactions image to Docker registry, provide configuration information, and then install MicroTx Distributed Transactions.	You can install MicroTx Distributed Transactions on a Kubernetes cluster or Docker Swarm. Provide configuration information in the <code>values.yaml</code> file for Kubernetes and the <code>tcs-docker-swarm.yaml</code> file for Docker Swarm.	Install on Kubernetes Cluster or Install on Docker Swarm
Access MicroTx Distributed Transactions	Verify that MicroTx Distributed Transactions was installed properly and access the service.	Access MicroTx
Run Sample Applications	Optional. Using samples is the fastest way for you to get familiar with MicroTx Distributed Transactions.	Deploy Sample Applications
Use MicroTx Distributed Transactions library with your application code.	Perform this step for all the transaction participant and transaction initiator applications so that your applications can access the library which interacts with MicroTx Distributed Transactions.	Perform this task based on the transaction protocol that you want to use. • Develop Applications with XA • Develop Applications with Saga • Develop Applications with TCC
Install and run your application	After using the library files in your application, install and run your applications.	Deploy Your Application

2.2 Access MicroTx Distributed Transactions

To access MicroTx Distributed Transactions, specify the port number, host name, and protocol that you want to use to access. Oracle recommends that you use HTTP protocol only in test or development environments. In production environments, you must use HTTPS protocol.

Use the internal URL or external URL to access MicroTx Distributed Transactions. You will use different URLs depending on whether you want to access MicroTx Distributed Transactions from within the Kubernetes cluster where you have deployed the service or from a different Kubernetes cluster.

Internal URL to access MicroTx Distributed Transactions

Use the internal URL to access MicroTx Distributed Transactions from within the Kubernetes cluster where you have deployed the service. For example, when you have deployed the transaction initiator application and MicroTx Distributed Transactions in the same Kubernetes cluster.

To access MicroTx Distributed Transactions, create the URL in the following format:

```
http://internalHostname:internalPort/api/v1
```

Where,

- *internalHostname*: Name that you have entered for the `tmmAppName` property in the `values.yaml` of the MicroTx Distributed Transactions. For example, `tmm-app`.
- *internalPort*: Port number that you have entered for the `port` property in the `values.yaml` of the MicroTx Distributed Transactions. For example, `9090`. Ensure that you have set up the required networking rules to permit HTTPS traffic on this port.

Based on the example values provided above, the example MicroTx Distributed Transactions URL is `https://tmm-app:9090/api/v1`.

All communication within a container uses the HTTP protocol as the communication goes through the Envoy proxy, which uses mTLS.

External URL to access MicroTx Distributed Transactions

Use the external URL to access MicroTx Distributed Transactions from outside the Kubernetes cluster where you have deployed the service. For example, when you deploy the transaction initiator application and MicroTx Distributed Transactions in different Kubernetes cluster. In such a scenario, the transaction initiator application uses the external URL to access MicroTx Distributed Transactions.

To access MicroTx Distributed Transactions externally, create the URL in the following format:

```
https://externalHostname:externalPort/api/v1
```

Where,

- `https` is the protocol permitted at the Istio ingress gateway. You can specify either `http` or `https`.
- *externalHostname*: The IP address of the load balancer of the Istio ingress gateway. See [Find IP Address of Istio Ingress Gateway](#). For example, `192.0.2.1`.

- *externalPort*: Port number of the load balancer of the Istio ingress gateway. You must create the required networking rules to permit inbound and outbound traffic on this port. For example, 443.

Based on the example values provided above, the example MicroTx Distributed Transactions URL is `https://192.0.2.1:443/api/v1`.

2.3 Find IP Address of Istio Ingress Gateway

Before you start a transaction, you must note down the external IP address of the Istio ingress gateway.

You need this information to access the applications.

1. Run the following command to find the external IP address of the Istio ingress gateway.

Command

```
kubectl get svc istio-ingressgateway -n istio-system
```

Sample Output

```
kubectl get svc istio-ingressgateway -n istio-system
NAME                                TYPE                CLUSTER-IP      EXTERNAL-IP
PORT(S)                              AGE
istio-ingressgateway    LoadBalancer    10.109.1.101     192.0.2.1
15021:31695/TCP,80:32333/TCP,443:7777/TCP    44h
```

2. From the output note down the value of EXTERNAL-IP, which is the external IP address of the Istio ingress gateway, and the port associated with the HTTP or HTTPS traffic, based on the access protocol that you have configured. For example: `https://192.0.2.1:443`.
3. Store the external IP address of the Istio ingress gateway in an environment variable named `CLUSTER_IPADDR` as shown in the following command.

```
export CLUSTER_IPADDR=192.0.2.1
```

Note that, if you don't do this, then you must explicitly specify the IP address in the commands when required.

2.4 About MicroTx Distributed Transactions Client Libraries

After installing MicroTx Distributed Transactions, you can integrate MicroTx Distributed Transactions client libraries with your application code or run sample applications to understand how MicroTx Distributed Transactions works.

The MicroTx Distributed Transactions client libraries are available at <https://mvnrepository.com/artifact/com.oracle.microtx>.

To integrate the MicroTx client libraries with your application code, refer to one of the following topics based on the transaction protocol that your applications use.

- [Develop Applications with XA](#)
- [Develop Applications with Saga](#)
- [Develop Applications with TCC](#)

You can also run sample applications to understand how MicroTx works. Using sample applications is the fastest way for you to get familiar with MicroTx. See [Deploy Sample Applications](#).

3

Develop Applications with XA

To use Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions) to manage the transactions of your microservices, you need to make a few changes to your existing application code to integrate the functionality provided by the MicroTx Distributed Transactions libraries.

This section provides instructions to integrate MicroTx Distributed Transactions library with your application. Refer to the sample code of Bank Transfer application that uses the XA transaction protocol to transfer an amount from one department to another. The MicroTx Distributed Transactions library files are already integrated with the application code. Link to the relevant files is available in each section. Use these files as a reference when you integrate the MicroTx Distributed Transactions libraries with your application.

Topics

- [About MicroTx Distributed Transactions Library for XA](#)
To use Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions) to manage the transactions of your microservices, you need to make a few changes to your existing application code to integrate the functionality provided by the MicroTx Distributed Transactions libraries.
- [Workflow to Develop Applications with XA](#)
Use the following workflow as a guide to develop your applications to use MicroTx Distributed Transactions to manage XA transactions.
- [Plan Your Resource Manager](#)
Consider the points discussed in this section to plan the resource manager. Based on the resource manager that you select and how you use it, the configuration requirements vary for your application.
- [Provide the Resource Manager Connection Details](#)
For every transaction participant application that uses a resource manager to store the application data, enter the connect string, user name, and password to access the resource manager in the application's YAML file. Skip this section if your application does not use a resource manager.
- [Configure PostgreSQL as Resource Manager](#)
To use PostgreSQL as resource manager for XA transactions, you must enable prepared transactions and session affinity.
- [Manage XA Transactions](#)
- [Required User Privileges](#)
You must have the following privileges to execute an XA transaction.
- [Configure Library Properties for JAX-RS and Node.js Apps](#)
Provide configuration information for the MicroTx Distributed Transactions library properties for JAX-RS and Node.js applications. Specify the library property values for both participant and initiator applications.
- [About @Transactional](#)
You can configure your Java applications that use the XA transaction protocol in two distinct ways.

- [Develop Spring Boot Apps with XA](#)
Use the MicroTx Distributed Transactions library with your Spring Boot applications.
- [Develop JAX-RS Apps with XA](#)
This section provides information about using the MicroTx Distributed Transactions library with Helidon applications that use the JAX-RS implementation.
- [Configure Java Apps to Leverage Transactional Event Queues](#)
Integrate Oracle MicroTx Distributed Transactions library with your application to leverage the Oracle Transactional Event Queues (TEQ) feature.
- [Develop Node.js Apps with XA](#)
- [Develop ORDS App with XA](#)
Configure an Oracle Database application, that you have built using Oracle APEX and Oracle REST Data Services (ORDS), as a transaction initiator or participant in an XA transaction with MicroTx Distributed Transactions.
- [Include Oracle Blockchain Platform in XA Transactions](#)
You can use MicroTx to manage XA transactions in Oracle Blockchain Platform for Java applications, such as Spring Boot and Helidon.

3.1 About MicroTx Distributed Transactions Library for XA

To use Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions) to manage the transactions of your microservices, you need to make a few changes to your existing application code to integrate the functionality provided by the MicroTx Distributed Transactions libraries.

1. Include the MicroTx Distributed Transactions client libraries in your microservice implementation.
2. Use CDI annotations or MicroTx Distributed Transactions client libraries APIs to register the required interceptors and callbacks.
3. Use CDI annotations or MicroTx Distributed Transactions client library APIs in participant microservices to obtain the connection to their XA compliant resource manager.
4. Use MicroTx Distributed Transactions client libraries API to delineate transaction boundaries indicating an XA transaction has started, and then commit or roll back the transaction.

The MicroTx Distributed Transactions library is available for Java, Node.js, ORDS, Tuxedo, and WebLogic Server apps.

3.2 Workflow to Develop Applications with XA

Use the following workflow as a guide to develop your applications to use MicroTx Distributed Transactions to manage XA transactions.

Task	Description	More Information
Install MicroTx Distributed Transactions	Install MicroTx Distributed Transactions and ensure that you can access it.	Workflow to Install MicroTx
Set up resource manager for your transaction participant applications	Identify the type of resource manager that you want to use, such as XA-compliant or non-XA compliant.	Plan Your Resource Manager

Task	Description	More Information
Provide configuration information for the MicroTx Distributed Transactions library properties.	Perform this step for all the transaction participant and transaction initiator applications so that your applications can access the library.	Configure Library Properties for JAX-RS and Node.js Apps
Integrate MicroTx Distributed Transactions library with your application code.	Select a suitable procedure to integrate the library based on the following factors: - the development framework for your application - whether an application initiates the transaction or participates in the transaction	Based on your app, perform one of the following tasks: Develop Spring Boot Apps with XA Develop JAX-RS Apps with XA Develop Node.js Apps with XA Configure JPA-Based Java App as Transaction Participant Develop ORDS App with XA Develop Tuxedo Apps with XA
Deploy your application	Develop, test, and deploy your microservices independently. After using the library files in your application, the application in your environment.	Deploy Your Application

3.3 Plan Your Resource Manager

Consider the points discussed in this section to plan the resource manager. Based on the resource manager that you select and how you use it, the configuration requirements vary for your application.

- [Supported Resource Managers](#)
The transaction participant services may use a resource manager to store application data.
- [Supported Drivers for Resource Managers](#)
It is the application developer's responsibility to select the correct JDBC driver and UCP version, if required, that works with the resource manager that you want to use.
- [Optimizations for a Non-XA Resource](#)
Use the Logging Last Resource (LLR) or Last Resource Commit (LRC) optimization to enable one non-XA resource to participate in a global transaction.
- [Common Resource Manager for Multiple Apps](#)
When you use a common resource manager for multiple participant services, MicroTx can optimize the commit processing resulting in higher throughput and lower latency for XA transactions.
- [Configure Multiple Resource Managers for a Single App](#)
You can use multiple resource managers for a single participant service. Based on the business logic, a participant service can connect to multiple XA-compliant resource managers. However, only one non-XA resource is supported in a transaction.
- [About Dynamic Recovery for XA Transactions](#)
The transaction coordinator server resumes the transactions that were in progress when the server restarts after a failure.

3.3.1 Supported Resource Managers

The transaction participant services may use a resource manager to store application data.

In XA transactions, the MicroTx Distributed Transactions libraries need to access the resource manager's client libraries.

Supported Resource Managers for Java Applications

For Java applications that participate in an XA transaction, the MicroTx Distributed Transactions library is tested with the following resource managers:

- Oracle Database 19c
- Oracle Real Application Clusters (RAC) 19c
- PostgreSQL 14.2

Supported Resource Managers for Node.js Applications

For Node.js applications that participate in an XA transaction, the MicroTx Distributed Transactions library is tested with Oracle Database 19.x and Oracle RAC 19c.

Supported Non-XA Resources

An application participating in an XA transaction can use a single non-XA resource. MicroTx Distributed Transactions library is tested with the following non-XA resources:

- MongoDB 4.1 or later
- MySQL

For more information, see [Optimizations for a Non-XA Resource](#).

Supported Databases for Applications Using TCC or Saga Transaction Protocol

If you select Saga or TCC as the transaction protocol, you can use any database to store your application data. The MicroTx Distributed Transactions library does not interact with the application database in Saga and TCC transaction protocols.

3.3.2 Supported Drivers for Resource Managers

It is the application developer's responsibility to select the correct JDBC driver and UCP version, if required, that works with the resource manager that you want to use.

Working with Oracle Database as resource manager

You must use a supported JDBC driver and UCP version that works with Oracle Database. The MicroTx Distributed Transactions library accesses the XAResource object to perform various XA operations on the resource manager. This XAResource object is provided by the JDBC driver.

For the MicroTx Distributed Transactions Java library, Universal Connection Pool (UCP) is used along with the Oracle JDBC driver for improved performance.

The MicroTx Distributed Transactions libraries for Java is tested with Oracle Database drivers version 21.3.0.0.

If you use Oracle Database as the resource manager and MicroTx Distributed Transactions Node.js library, you must use node-oracledb 6.1.0 in the participant application.

There are no additional requirements for database drivers if you are using Logging Last Resource (LLR) transactions.

Working with resource managers other than Oracle Database

You must use a supported JDBC driver that implements the XADataSource and XAResource interfaces. The MicroTx Distributed Transactions library accesses the XAResource object to perform various XA operations on the resource manager.

3.3.3 Optimizations for a Non-XA Resource

Use the Logging Last Resource (LLR) or Last Resource Commit (LRC) optimization to enable one non-XA resource to participate in a global transaction.

Your microservice may contain several participant applications, where each application may be connected to a different resource manager. For example, a microservice contains a transaction initiator application which uses Oracle Database as the resource manager and a transaction participant application which uses MongoDB as the resource manager. MongoDB does not support the XA protocol. However, both MongoDB and Oracle Database need to participate in a global transaction. With MicroTx Distributed Transactions, you can use the XA transaction protocol for such a microservice when you enable LLR or LRC optimization.

About Logging Last Resource (LLR) Optimization

Use the LLR optimization to enable one non-XA resource to participate in a global transaction *with* the same ACID guarantee as XA.

XA resources can handle the XA requests sent by the transaction coordinator, such as prepare, commit, and rollback. Non-native or non-XA resources cannot handle such requests. The LLR and LRC optimizations enable a single non-XA resource to participate in an XA transaction. The transaction coordinator prepares all the other branches of the transaction, and then attempts to perform a local transaction commit to the LLR or LRC branch. Assuming that all the other branches are prepared without an issue, the outcome of the local commit determines the outcome of the transaction. If the local commit takes place successfully, the transaction is committed successfully, otherwise the transaction is rolled back.

Before performing a local commit, the transaction coordinator creates a commit record in the LLR branch. In case of any failure, the transaction coordinator tries to recover the list of transactions by calling `xa_recover` on the LLR branch. If the LLR branch had successfully committed its local transaction, the `commitRecord` returns the list of prepared participants. If the LLR branch failed to commit its local transaction, the `recover()` method returns an indication that no participants were recorded.

By default, the commit records are deleted after two hours. You can modify the time period for which the commit records are retained by specifying the time period in the `oracle.tmm.LlrDeleteCommitRecordInterval` property. For example, if you specify 600000 ms or 10 minutes, then only the commit records that were created during the last 10 minutes are retained and the previous records are deleted.

If the LLR branch succeeds in committing the local transaction that also includes the commit record for the transaction coordinator, then `recover()` returns the commit record.

About Last Resource Commit (LRC) Optimization

You can use the LRC optimization to enable one non-XA resource to participate in a global transaction *without* the same ACID guarantee as XA.

In LRC, the sequence of flow of the transaction is nearly identical to LLR. When the initiator calls `commit` on the transaction coordinator, the transaction coordinator prepares all the XA branches, and then calls `commit()` on the LRC branch. The only difference is that you can't recover the transaction details in case of any failure as the `commit()` method returns `NULL` as the value for `commitRecord` in LRC. In LLR, the `commit()` method returns a list of prepared participants in response. When `commit()` is called in LRC, the local transaction is committed and the outcome is returned to the transaction coordinator, but information about the prepared participants is not stored.

As information about the transaction log details is not stored, LRC optimization works with all supported resource managers. However, the possibility of heuristic outcomes increases as there is no way for the transaction coordinator to check if the local commit was completed successfully. Also, you can't use the `recover()` methods in LRC, so you can't recover the transaction in case of any failure.

Choose between LLR and LRC

Oracle strongly recommends that you use the LLR optimization for your non-XA resource as you can recover details in case of a failure. Use the LRC optimization only when your non-XA resource cannot store the `commitRecord` details or transaction log details.

Limitations

- MicroTx Distributed Transactions supports only *one* participant application with a non-XA resource to participate in XA transactions with LLR or LRC optimization. If your microservice has multiple non-XA resources, then MicroTx Distributed Transactions does not support the XA transaction protocol for this microservice. For example, the following error message is displayed if you try to use multiple LLR or LRC participants: Only one LLR or LRC participant is allowed to enlist.

If the initiator application participates in the transaction after starting the transaction, then you can use an LLR or LRC resource with this initiator application.

3.3.4 Common Resource Manager for Multiple Apps

When you use a common resource manager for multiple participant services, MicroTx can optimize the commit processing resulting in higher throughput and lower latency for XA transactions.

When you use a common resource manager for multiple participant services, you can specify a value for the `ORACLE_TMM_XA_RMID` environment variable to optimize the transaction. The transaction is optimized as only one branch is created for all the participant services that share a resource manager.

Let us consider that Dept A, Dept B, and Dept C are three participant services that share a resource manager, but have different `ORACLE_TMM_XA_RMID` values. MicroTx Distributed Transactions creates a new branch for each department. In all MicroTx Distributed Transactions creates three branches to track the transactions.

To optimize the transaction, specify a unique value, such as `ORCL1`, for the `ORACLE_TMM_XA_RMID` environment variable in the Dept A, Dept B, and Dept C services.

When you specify a value for the `ORACLE_TMM_XA_RMID` environment variable, MicroTx Distributed Transactions creates a single branch for all the services that use a single resource manager. Since multiple branches are not created, the transaction is optimized. In this scenario, MicroTx Distributed Transactions optimizes the transaction and creates a single branch to track the transactions that involve the common resource manager and multiple participants. When you don't provide a value for this variable, MicroTx Distributed Transactions does not optimize the transaction and creates three branches, one for each participant service.

Note

If you use the Oracle RAC database as a common resource manager for multiple participant services, you *must* specify the same RMID value for all the participant services that use a common Oracle RAC database as resource manager.

Limitations

- You can only share an XA-compliant resource manager with multiple participant services. You cannot share a non-XA resource with multiple participants services.
- You can use a common resource manager for all transaction participant services, including an initiator application which participates in the transaction. A transaction initiator service, which initiates the transaction but does not participate in the transaction, does not require a resource manager.
- You must use unique RMIDs for different resource managers. The transaction fails if you use same RMID for different resource managers.

3.3.5 Configure Multiple Resource Managers for a Single App

You can use multiple resource managers for a single participant service. Based on the business logic, a participant service can connect to multiple XA-compliant resource managers. However, only one non-XA resource is supported in a transaction.

Note

This feature is available only in the MicroTx Distributed Transactions client libraries for Java applications. JPA or Hibernate applications support only XA-compliant resource managers.

3.3.6 About Dynamic Recovery for XA Transactions

The transaction coordinator server resumes the transactions that were in progress when the server restarts after a failure.

Every time transaction coordinator restarts, it recovers transactions for all protocols (XA, Saga, and TCC) based on the data available in the transaction store. See [About Transaction Recovery](#).

Additionally, for XA transaction protocol, the transaction coordinator dynamically recovers the transactions which are not committed. The transaction coordinator checks for any transactions that were in progress when the coordinator failed, then the coordinator issues a commit or roll back command to complete the transaction. If the transaction is not found or it has already been completed, then the coordinator removes the transaction record from the resource manager.

Dynamic recovery is performed based on the resource manager ID (RMID) that you specify. Ensure that the RMID that you specify for each resource manager is unique.

The transaction coordinator performs dynamic recovery once for each resource manager based on the RMID. If the transaction coordinator instance restarts, then the recovered information is not lost but the mapping of the recovered RMID list is lost. During dynamic recovery `xa_recover` is called once for every RMID. The recovered information about the resource manager is kept in memory. Every time participants enlist, the transaction coordinator checks the RMID against recovered resource manager mapping which is kept in memory. This ensures that only if an RMID does not exist in the already recovered list, then `xa_recover` is called. If the RMID exists in the recovered list, `xa_recover` is not called. Since the resource manager mapping is kept in memory, if the transaction coordinator restarts, the list of recovered RMID list is lost. In such a scenario, the recovery is called again when each unique RMID enlists.

If you have set up etcd or Oracle Database for MicroTx Distributed Transactions to store the transaction data, then you can obtain information about the in-progress transactions and transaction details after the coordinator restarts. However, if you haven't set up a separate transaction store and are using internal memory to store the transaction details, then all the stored information is lost after the coordinator crashes or restarts. Since XA supports dynamic recovery, all the dynamically recovered (xa_recover) XA transactions are rolled back and followed by xa_forget in case you are using internal memory.

3.4 Provide the Resource Manager Connection Details

For every transaction participant application that uses a resource manager to store the application data, enter the connect string, user name, and password to access the resource manager in the application's YAML file. Skip this section if your application does not use a resource manager.

Enter the connection string to access the resource manager as the value for the databaseUrl property. The format in which you provide the connection string depends on the type of the database that you use and the language of the application's source code. Connection strings are provided in this section only for your reference. Refer to the database-specific documentation for the latest information.

Format of the Connection String for Java Applications

For all Java applications, such as JAX-RS, Spring-REST, Hibernate, or EclipseLink, enter the connection string in one of the following formats depending on the type of the database.

- If you are using a database that uses a credential wallet, such as Oracle Autonomous Transaction Processing, use the following format to enter the connection string:

```
jdbc:oracle:thin:@tcps://<host>:<port>/<service_name>?  
wallet_location=<wallet_dir>
```

You can find the required details, such as protocol, host, port, and service name in the tnsnames.ora file, which is located in folder where you have extracted the wallet. See [Download Client Credentials \(Wallets\)](#) in *Using Oracle Autonomous Database on Shared Exadata Infrastructure*.

For example:

```
jdbc:oracle:thin:@tcps://adb.us-phoenix-1.oraclecloud.com:1521/  
sales.adb.oraclecloud.com?wallet_location=Database_Wallet
```

- If you are using a non-autonomous Oracle Database, a database that does not use a credential wallet, use the following format to enter the connection string:

```
jdbc:oracle:thin:@<publicIP>:<portNumber>/  
<databaseUniqueName>.<hostDomainName>
```

For example:

```
jdbc:oracle:thin:@123.213.85.123:1521/  
CustDB_iad1vm.sub05031027070.customervcnwith.oraclevcn.com
```

Where, CustDB_iad1vm.sub05031027070.customervcnwith is the unique name of the database or the service name.

- If you are using Oracle Database Cloud Service with Oracle Cloud Infrastructure, see [Create the Oracle Database Classic Cloud Service Connection String](#) in *Using Oracle Blockchain Platform*.
- If you are using Oracle RAC database, see [About Connecting to an Oracle RAC Database Using SCANS](#).
- If you are using MySQL database, use the following format to enter the connection string:

```
jdbc:mysql://<database-host>:<database-port>/<database-name>
```

For example:

```
jdbc:mysql://10.0.10.59:3306/department_nonxa_ds
```

For more information, refer to [MySQL documentation](#).

- If you are using PostgreSQL database, use the following format to enter the connection string:

```
jdbc:postgresql://<postgres-host>:<postgres-port>/<database-name>
```

For example:

```
jdbc:postgresql://postgres-deployment-  
postgresql.postgres.svc.cluster.local:5432/department
```

For more information, refer to [PostgreSQL documentation](#).

- If you are using MongoDB database, use the following format to enter the connection string:

```
mongodb://<mongodb-host>:<mongodb-port>/?replicaSet=rs0
```

For details, refer to the [MongoDB documentation](#).

Format of the Connection String for Node.js Applications

For Node.js XA transaction participant applications, MicroTx Distributed Transactions library is tested with Oracle Database 19.x and Oracle RAC 19c.

If you are establishing a connection between a Node.js application and an Oracle database, provide the connection string in the format specified at [Embedded Connect Descriptor Strings](#).

3.5 Configure PostgreSQL as Resource Manager

To use PostgreSQL as resource manager for XA transactions, you must enable prepared transactions and session affinity.

Skip this section if you don't want to use PostgreSQL as a resource manager.

By default, the value of `max_prepared_transactions` is set to 0 and prepared transactions are disabled. If you do not enable prepared transactions for PostgreSQL, you will receive the following error message when you start an XA transaction.

Exception: `org.postgresql.util.PSQLException: ERROR: prepared transactions are disabled`

1. Add the following dependency to your application's `pom.xml` file.

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>RELEASE</version>
</dependency>
```

2. Connect to the database using pgAdmin 4 or any another PostgreSQL tool, and then run the following SQL statements. Set the value of `max_prepared_transactions` to a positive number, such as 100, to enable prepared transactions for PostgreSQL.

```
SHOW max_prepared_transactions;
ALTER SYSTEM SET max_prepared_transactions = 100;
```

3. Restart the PostgreSQL service.

```
services restart postgresql
```

4. Enable session affinity or sticky sessions for the transaction participant service that uses PostgreSQL as resource manager. When you enable session affinity, all the requests for a unique transaction or session are routed to the same endpoint or replica of the participant service that served the first request. See [Enable Session Affinity](#).

3.6 Manage XA Transactions

- [Set Transaction Timeout](#)
Specify the time period for which a request sent from the XA participant services remains active. If a transaction is not committed or rolled back within the specified time period, the transaction is rolled back.
- [About Global and Local Transactions](#)
When a transaction initiator service makes a call to MicroTx Distributed Transactions to begin an XA transaction, by default this transaction is handled as a global transaction.
- [About Asynchronous Calls to Participant Apps](#)
The MicroTx Distributed Transactions library supports asynchronous calls from an XA transaction initiator application to participant applications. This improves performance when the initiator must call multiple participant services, because the calls can run concurrently instead of sequentially.
- [Subscribe to Receive XA Transaction Notifications](#)
You can register your transaction initiator and participant services to receive notifications. MicroTx Distributed Transactions notifies the registered services when the following events occur: before the prepare phase and when MicroTx Distributed Transactions successfully commits or rolls back a transaction.

3.6.1 Set Transaction Timeout

Specify the time period for which a request sent from the XA participant services remains active. If a transaction is not committed or rolled back within the specified time period, the transaction is rolled back.

Specify this value only for the transaction initiator application. The transaction initiator service propagates the timeout value, that you have specified, to the participant services. If you specify this value for a participant application, it is ignored.

Note

The value that you provide specifies the timeout for the overall transaction. While deciding the transaction timeout, you must consider the resource manager latency too.

To set transaction timeout for requests sent from participants services:

1. In the transaction coordinator configuration parameters, specify the *maximum* amount of time, in milliseconds, for which a transaction remains active. Specify a value for the `txMaxTimeout` parameter in the `values.yaml` file of the MicroTx Distributed Transactions coordinator. The default value is 60000 ms.

The `values.yaml` file of the MicroTx Distributed Transactions is located in the `installation_directory/MicroTx-RELEASE/MicroTx/helmcharts` folder.

2. In the transaction initiator service, specify the amount of time, in milliseconds, for which a transaction remains active. If a transaction is not committed or rolled back within the specified time period, the transaction is rolled back. The default value and minimum value is 60000. You can specify this value in the application code or as a property value. The value specified in the application code overrides the property value.

Specify a value for one of the following MicroTx Distributed Transactions library properties, depending on the type of your initiator service.

- For Node.js and JAX-RS applications based on Helidon, provide a value for the `oracle.tmm.TransactionTimeout` property.
- For Spring Boot applications, provide a value for the `spring.microtx.xa-transaction-timeout` property.

The value of transaction timeout specified for the initiator service can override the value of `txMaxTimeout` specified for the transaction coordinator, but it cannot exceed the value of `txMaxTimeout`. For example, if the value of `txMaxTimeout` is 70000 and the value of transaction timeout specified for the initiator service is 80000, then the maximum timeout is set to 70000 milliseconds. If the value of `txMaxTimeout` is 90000 and the value of transaction timeout specified for the initiator service is 80000, then the maximum timeout is set to 80000 milliseconds.

3.6.2 About Global and Local Transactions

When a transaction initiator service makes a call to MicroTx Distributed Transactions to begin an XA transaction, by default this transaction is handled as a global transaction.

Global transaction is a transaction that is associated with a GTRID. For example, transactions that span across multiple microservices or transactions that involve a single microservice interacting with multiple resource managers.

Local transaction is a transaction that is not associated with a GTRID. For example, when a transaction includes a single service that interacts with a single resource manager. Such transactions can be handled locally, as you don't need a coordinator to manage this transaction. The MicroTx Distributed Transactions client library manages the local transactions. By handling a transaction locally, without using the MicroTx Distributed Transactions coordinator, you can experience better performance. Local transactions save time and increase throughput.

- [About MicroTx Distributed Transactions Client Library Annotations](#)
Based on the context in which a participant service is called, the service participates in a local or global XA transaction.
- [About Transaction Promotion](#)
When you enable the transaction promotion feature, every transaction is initiated as a local transaction.
- [Enable Transaction Promotion](#)
By default, all transactions are associated with a global transaction ID (GTRID) and handled as a global transaction. To experience better performance, enable transaction promotion to manage certain transactions.

3.6.2.1 About MicroTx Distributed Transactions Client Library Annotations

Based on the context in which a participant service is called, the service participates in a local or global XA transaction.

The following annotations, which are part of the MicroTx Distributed Transactions client library, create a local transaction or a global XA transaction based on the context in which a participant service REST API is called.

- For JAX-RS applications: `@TrmSQLConnection`, `@TrmEntityManager`, and `@TrmNonXASQLConnection` annotations.
- For Spring REST applications: `@microTxSqlConnection`, `@microTxEntityManager`, and `@microTxNonXASqlConnection` annotations.

When a participant service is called in the context of a global XA transaction, the MicroTx Distributed Transactions coordinator provides connections as part of the global XA transaction.

In a local transaction, a participant service does not interact with any other service or the MicroTx Distributed Transactions coordinator. The MicroTx Distributed Transactions coordinator does not callback a participant service which is not a part of a global transaction or when a participant service is called in the context of a local transaction. Application developers are responsible for handling commits and other operations for local transactions according to their business requirements.

3.6.2.2 About Transaction Promotion

When you enable the transaction promotion feature, every transaction is initiated as a local transaction.

Note

This feature is available only in Oracle Database.

Some transactions are handled locally and are not promoted as a global transaction. For example, when the transaction initiator service interacts with only one resource manager and does not communicate with any external service, the transactions are handled locally.

The following transactions are promoted as global transactions as they may not be handled locally.

- When the transaction initiator service communicates with multiple resource managers.
- When multiple services, along with their resource managers, participate in a transaction.

If the transaction is promoted as a global transaction, then the service enlists with the MicroTx coordinator and the coordinator handles the transaction.

3.6.2.3 Enable Transaction Promotion

By default, all transactions are associated with a global transaction ID (GTRID) and handled as a global transaction. To experience better performance, enable transaction promotion to manage certain transactions.

Note

This feature is available only in Oracle Database.

To enable transaction promotion:

- Set the `oracle.tmm.promotable` property to `true` in the `tmm.properties` file for the initiator service. When you enable transaction promotion, the transactions which can be handled locally are managed without involving the coordinator. The default value is `false`. If you set the value of the `oracle.tmm.promotable` property as `false`, every transaction starts as a global transaction by default and it is associated with a GTRID. A local transaction is not initiated.

3.6.3 About Asynchronous Calls to Participant Apps

The MicroTx Distributed Transactions library supports asynchronous calls from an XA transaction initiator application to participant applications. This improves performance when the initiator must call multiple participant services, because the calls can run concurrently instead of sequentially.

This feature is available for Helidon and Spring Boot applications that participate in XA transactions.

How it works

MicroTx stores XA transaction context in thread-local state. When an application uses `ExecutorService`, `FutureTask`, `CompletableFuture`, `Spring @Async`, or another asynchronous execution mechanism, the new worker thread does not automatically inherit the caller thread's MicroTx transaction context.

To use asynchronous XA calls, the application must use the MicroTx-provided async context propagation support:

- For Spring Boot, configure the async executor with `MicroTxXAContextTaskDecorator`.
- For Helidon, use `MicroTxXATaskExecutor` or an equivalent executor that propagates the MicroTx XA context.

The initiator must wait for all asynchronous participant calls to finish before calling commit or rollback. If any asynchronous participant call fails, the initiator must roll back the XA transaction.

Participant limit

MicroTx does not define a fixed hard limit on the number of participants that can be called asynchronously.

The practical limit depends on:

- Executor thread-pool size
- HTTP client connection-pool size
- Participant service capacity
- Resource manager capacity
- Transaction coordinator capacity
- Transaction timeout- CPU, memory, and network resources

Configure the executor and connection pools based on the expected number of participants and concurrent transactions.

Supported use cases

MicroTx supports asynchronous calls to participant services in the following XA use cases:

- The transaction initiator calls multiple participant applications concurrently.
- The transaction initiator propagates the same XA transaction context to each asynchronous call.
- Each participant enlists correctly with the MicroTx coordinator.
- The initiator waits for all participant calls to complete before commit or rollback.
- The initiator application does not perform concurrent asynchronous work on the same local XA resource context.
- The transaction promotion feature is not enabled.

This applies to both pure initiator applications and initiator-as-participant applications, provided the initiator-side XA work is not performed concurrently with participant calls in a way that reuses the same XA connection or resource context unsafely.

Unsupported or restricted use cases

MicroTx does not support, or does not reliably support, asynchronous XA participant calls in the following cases:

- Transaction promotion is enabled, where a local transaction is promoted to a distributed XA transaction.
- The initiator application uses multiple local resource managers while also making asynchronous participant calls.
- Multiple asynchronous threads use the same XA connection, same XA branch, or same resource manager context concurrently.
- The initiator commits or rolls back before all asynchronous participant calls have completed.
- The transaction includes non-XA participants or last-resource optimization participants.

- A common resource manager is shared by multiple async branches unless the resource manager, JDBC driver, and application access pattern explicitly support that concurrent XA usage.

Implementation references

You can use the following files as a reference while integrating the MicroTx libraries with your application.

- `TransferResourceAsync.java`: a sample transaction initiator application, which uses the MicroTx Distributed Transactions library to make asynchronous calls to participant applications in an XA transaction. See the [teller-as-participant](#) folder which is located in the `microtx-samples` GitHub repository.
- `TransferResourceWithSpringAsync.java`: Spring Boot initiator using Spring `@Async`. See the [teller-as-participant-spring](#) folder which is located in the `microtx-samples` GitHub repository.
- `TransferResourceWithFutureTaskAsync.java`: Spring Boot initiator using `FutureTask` and `ExecutorService`. See the [teller-as-participant-spring](#) folder which is located in the `microtx-samples` GitHub repository.

3.6.4 Subscribe to Receive XA Transaction Notifications

You can register your transaction initiator and participant services to receive notifications. MicroTx Distributed Transactions notifies the registered services when the following events occur: before the prepare phase and when MicroTx Distributed Transactions successfully commits or rolls back a transaction.

The MicroTx Distributed Transactions coordinator notifies the services that you register. You may want to register your service, if based on the business logic your service performs additional tasks when an event occurs. For every resource that you register, you must create a callback resource and declare two methods which MicroTx Distributed Transactions calls to send the notification when an event occurs.

Note

This feature is available only for Java services.

Perform the following task for the transaction participant and initiator services that you want to register to receive event notifications.

1. Within your application code, add code to create a callback resource that the MicroTx Distributed Transactions coordinator can call when an event occurs.

Create a JAX-RS class with two methods. It is mandatory for you to declare the `beforeCompletion` and `afterCompletion` methods. Within these methods, provide code that is specific to your application's business logic. The MicroTx Distributed Transactions coordinator calls the `beforeCompletion` method before sending a request to the participants to prepare. The `afterCompletion` method returns the final status of the event after the transaction is complete. The status of the can be `STATUS_COMMITTED` or `STATUS_ROLLEDBACK`.

In the following sample code, `EventListenerResource` is the name of the JAX-RS class and `transaction-sync` is the name of the callback resource. You can provide any name of your choice for the class and callback resource. Note down the name of this resource as you will provide it later.

Sample code

```

@Path("transaction-sync")
public class EventListenerResource {

    /**
     * The MicroTx Distributed Transactions coordinator calls the
     beforeCompletion method before
     * the two-phase transaction commit process starts. This call is
     executed with
     * the transaction context of the transaction that is being committed.
     */
    @POST
    @Path("/{gtrid}/beforecompletion")
    @Produces(MediaType.APPLICATION_JSON)
    public Response beforeCompletion(@PathParam("gtrid") String gtrid) {
        ...
        //tasks to be done before the transaction is completed
        //enter the code based on your application's business logic
        return Response.status(Response.Status.OK).build();
    }

    /**
     * The MicroTx Distributed Transactions coordinator calls the
     afterCompletion method after the
     * transaction is committed or rolled back.
     */
    @POST
    @Path("/{gtrid}/aftercompletion/{status}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response afterCompletion(@PathParam("gtrid") String gtrid,
    @PathParam("status") String status) {
        ...
        //tasks to be done after the transaction is completed
        //enter the code based on your application's business logic
        return Response.status(Response.Status.OK).build();
    }
}

```

2. Register the initiator service to receive event notifications based on your application's business logic.

The following sample code describes that you call the `TrmRegisterSynchronization.register()` method after calling `begin()`, but before calling `commit()` or `rollback()`. When you call the `TrmRegisterSynchronization.register()` method, you must pass the name of the callback resource that you have created in the previous step.

Sample code

```

import oracle.tmm.jta.TrmUserTransaction;
/**
 * Initiator method which initiates the transaction
 */
transactionMethod() {
    TrmUserTransaction transaction = new TrmUserTransaction();
    transaction.begin();
}

```

```

        //
        TrmRegisterSynchronization.register(transaction.getTransactionID(), "/"
transaction-sync");
        ...
        // code that is specific to the application's business logic
        transaction.commit();
    }

```

Where,

- *transaction-sync* is the name of the callback resource that you have created in the previous step. Replace this value based on your environment.
 - `transaction.getTransactionID()` is the GTRID of the current transaction. Use the `TrmUserTransaction` class object to retrieve the GTRID of the current transaction.
3. Register one or more transaction participant services to receive event notifications. Based on your application's business logic, you can decide whether your application requires to receive event notifications.

The following sample code demonstrates how you can call the `TrmRegisterSynchronization.register()` method by explicitly passing the GTRID value and the name of the callback resource that you have previously created.

Sample code

```

import oracle.tmm.jta.TrmRegisterSynchronization;
/**
 * Participant method which is in transaction context.
 * Transaction event registration using GTRID
 */
participantMethod1(){
    TrmXaContext trmXaContext = ThreadLocalXaContext.get();
    if (trmXaContext != null) {
        String currentTransactionGTRID = new
String(trmXaContext.trmXid.getGlobalTransactionId());
        TrmRegisterSynchronization.register(currentTransactionGTRID, "/"
transaction-sync");
    }
    ...
    // code that is specific to the application's business logic
}

```

Where,

- *transaction-sync* is the name of the callback resource that you have previously created. Replace this value based on your environment.
- *currentTransactionGTRID* is the GTRID of the current transaction. To retrieve the GTRID of the current transaction from `ThreadLocal`, use `TrmXaContext`. This applies to transaction initiator as participant services as well.

3.7 Required User Privileges

You must have the following privileges to execute an XA transaction.

- GRANT SELECT ON SYS.DBA_PENDING_TRANSACTIONS TO *user*;
- GRANT SELECT ON SYS.DBA_2PC_PENDING TO *user*;
- GRANT EXECUTE ON SYS.DBMS_XA TO *user*;
- GRANT FORCE ANY TRANSACTION TO *user*;

Your system administrator or database administrator can grant these privileges.

3.8 Configure Library Properties for JAX-RS and Node.js Apps

Provide configuration information for the MicroTx Distributed Transactions library properties for JAX-RS and Node.js applications. Specify the library property values for both participant and initiator applications.

For JAX-RS and Node.js applications, open the `tmm.properties` file in any code editor, and then enter values for the following parameters to configure the MicroTx library.

- `oracle.tmm.TcsUrl`: Enter the URL to access the MicroTx Distributed Transactions application. See [Access MicroTx Distributed Transactions](#). You must enter this value for the transaction initiator application. You don't have to specify this value for the transaction participant applications.
- `oracle.tmm.TcsConnPoolSize`: Enter the number of connections to the MicroTx Distributed Transactions library to MicroTx Distributed Transactions. The default and minimum number of connections is 10. The maximum value is 20. You can change this value depending on the number of queries that your services run. Specify this value for both initiator and participant applications.
- `oracle.tmm.CallbackUrl`: Enter the URL of your initiator or participant service. MicroTx Distributed Transactions uses the URL that you provide to connect to the service. Provide this value in the following format:

`https://externalHostnameOfApp:externalPortOfApp/`

Where,

- `externalHostnameOfApp`: The external host name of your initiator or participant service. For example, `bookTicket-app`.
- `externalPortOfApp`: The port number over which you can access your initiator or participant service remotely. For example, `8081`.

You must specify this value for the transaction participant applications.

- `oracle.tmm.TransactionTimeout`: Specify the maximum amount of time, in milliseconds, for which the transaction remains active. If a transaction is not committed or rolled back within the specified time period, the transaction is rolled back. The default value and minimum value is 60000. Specify this value only for the transaction initiator application. The transaction initiator service propagates the timeout value, that you have specified, to the participant services. If you specify this value for a participant application, it is ignored. See [Set Transaction Timeout](#).

- `oracle.tmm.PropagateTraceHeaders`: Set this to `true` when you want to trace the transaction from end-to-end. This propagates the trace headers for all incoming and outgoing requests. For Helidon-based microservices, set this property to `false` to avoid propagating the trace headers twice as Helidon framework propagates trace headers by default. You can set this property to `true` if propagation of trace headers is disabled in Helidon configuration and you want to enable distributed tracing with MicroTx Distributed Transactions. For other microservices, set this property to `true`.
- `oracle.tmm.xa.Rmid`: Specify a unique string value for each resource manager that you use in the XA transaction. This value is not related to any properties of the data store. The unique value that you provide as RMID is used by MicroTx Distributed Transactions to identify the resource manager. If more than one participant uses the same resource manager, then specify the same resource manager ID for the participants that share a resource manager. It is mandatory to provide this value.
- `oracle.tmm.xa.XaSupport`: Set this to `true` when you use XA-compliant resources. Set this to `false` only for the single transaction participant service that uses a non-XA resource. The default value is `true`. When `oracle.tmm.xa.XaSupport` is set to `true`, the values set for `oracle.tmm.xa.LLRSupport` and `oracle.tmm.xa.LRCSupport` are ignored.
- `oracle.tmm.xa.LLRSupport`: Set this to `true` to enable the Logging Last Resource (LLR) optimization. Set this value only for the transaction participant service that uses a non-XA resource as a resource manager. The default value is `false`. When `oracle.tmm.xa.LLRSupport` is set to `true`, the value set for `oracle.tmm.xa.LRCSupport` is ignored.
Only when you set this property value to `true`, you can also specify a value for the `oracle.tmm.LlrDeleteCommitRecordInterval` property. Specify the maximum amount of time, in milliseconds, for which the committed records are retained. The default value is 7,200,000 ms or 2 hours. Before performing a local commit, the transaction coordinator creates a commit record in the LLR branch. The commit records are deleted after the time period specified in the `oracle.tmm.LlrDeleteCommitRecordInterval` property. Specify this value only for the service that uses a non-XA resource as a resource manager.
- `oracle.tmm.xa.LRCSupport`: Set this to `true` to enable the Last Resource Commit (LRC) optimization. Set this value only for the transaction participant service that uses a non-XA resource as a resource manager. The default value is `false`.
- `oracle.tmm.WeblogicTxnSupport`: Set this to `true` only for the JAX-RS applications, deployed on WebLogic Server, that participate in XA transaction. When you set this property to `true`, MicroTx Distributed Transactions ignores the values for `oracle.tmm.xa.XaSupport`, `oracle.tmm.xa.LLRSupport`, and `oracle.tmm.xa.LRCSupport`. The name of the corresponding environment variable is `ORACLE_TMM_WEBLOGIC_TXN_SUPPORT`.
- `oracle.tmm.promotable`: Set this to `true` to enable the transaction promotion feature. Specify this value only for a transaction initiator service that also participates in the transaction.
The default value is `false`. If you set the value of `oracle.tmm.promotable` as `false`, every transaction starts as a global transaction by default and it is associated with a GTRID. A local transaction is not initiated. See [About Global and Local Transactions](#).
- `oracle.tmm.xa.isRAC` or `oracle.tmm.xa.RACSupport`: Set this to `true` only for the transaction participant service that uses an Oracle Real Application Clusters (RAC) database as a resource manager. The default value is `false`. Specify this property value for Node.js applications, specify a value for the `oracle.tmm.xa.RACSupport` property. For JAX-RS applications, specify a value for the `oracle.tmm.xa.isRAC` property.

For example,

```
oracle.tmm.TcsUrl = http://tmm-app:9000/api/v1
oracle.tmm.TcsConnPoolSize = 15
oracle.tmm.CallbackUrl = https://bookTicket-app:8081
oracle.tmm.PropagateTraceHeaders = true
oracle.tmm.TransactionTimeout = 60000
oracle.tmm.xa.XaSupport = true
oracle.tmm.xa.LLRSupport = false
oracle.tmm.xa.LRCSupport = false
oracle.tmm.xa.Rmid = ORCL1
oracle.tmm.promotable = true
oracle.tmm.xa.isRAC = false
```

You can use the HTTP protocol if your application and MicroTx Distributed Transactions are in the same Kubernetes cluster, otherwise use the HTTPS protocol.

You can also provide these configuration values as environment variables. Note that if you specify values in both the properties file as well as the environment variables, then the values set in the environment variables override the values in the properties file.

The following example provides sample values to configure the environment variables.

```
export ORACLE_TMM_TCS_URL= http://tmm-app:9000/api/v1
export ORACLE_TMM_CALLBACK_URL = http://bookTicket-app:8081
export ORACLE_TMM_PROPAGATE_TRACE_HEADERS = true
export ORACLE_TMM_TCS_CONN_POOL_SIZE = 15
export ORACLE_TMM_TRANSACTION_TIMEOUT = 60000
export ORACLE_TMM_XA_XASUPPORT = true
export ORACLE_TMM_XA_LLRSUPPORT = false
export ORACLE_TMM_XA_LRC_SUPPORT = false
export ORACLE_TMM_XA_RMID = ORCL1
export ORACLE_TMM_XA_PROMOTABLE = true
export ORACLE_TMM_XA_ISRAC= false
```

Note that the environment variables names are case-sensitive.

3.9 About @Transactional

You can configure your Java applications that use the XA transaction protocol in two distinct ways.

- By defining transaction boundaries explicitly
- By using the @Transactional annotation

Each way offers its own advantages.

@Transactional Annotation

When you use the @Transactional annotation, you only need to provide the business logic. You don't need to define the transaction boundaries. Only if there is an exception, the transaction is rolled back. In all other scenarios, the transaction is committed.

The following code samples demonstrate the usage of the @Transactional annotation. Annotate the existing transfer method that defines the application business logic with @Transactional. The transfer method calls the withdraw method to debit an amount from

participant service 1 and the deposit method to credit an amount to participant service 2. Note that you only have to define the application's business logic in this case.

Sample Code for a JAX-RS Application Using @Transactional

```
import javax.transaction.Transactional;
...//some code
@Transactional(value = Transactional.TxType.REQUIRED)
transfer(){
    // code that is specific to the application's business logic
    // withdraw operation on participant service 1
    // deposit operation on participant service 2
}
```

For details about using @Transactional for JAX-RS apps, see <https://docs.oracle.com/javase/7/api/javax/transaction/Transactional.html>.

Sample Code for a Spring REST Application Using @Transactional

```
import org.springframework.transaction.annotation.Transactional;
...//some code
@Transactional(propagation = Propagation.REQUIRED)

transfer(){
    // code that is specific to the application's business logic
    // withdraw operation on participant service 1
    // deposit operation on participant service 2
}
```

For details about using @Transactional for Spring REST apps, see <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html>.

Define the Transaction Boundaries

When you want to specify certain conditions under which you want to commit or roll back a transaction, you can define the transaction boundaries. Use the `begin()` and `commit()` or `rollback()` methods to define the transaction boundaries. You must provide code to define the beginning of the transaction boundary and also provide the business logic to decide when to commit or rollback a transaction. This provides a lot of flexibility to define the scenarios in which the transaction is committed or rolled back.

The following samples shows the transfer method, which in turn calls the withdraw method to debit an amount from participant service 1 and the deposit method to credit an amount to participant service 2. Note that you only have to define the transaction boundaries and the application's business logic in this case.

Sample Code for a JAX-RS Application that Defines Transaction Boundaries

Create and handle the `TrmUserTransaction` object.

```
transfer() {
    //Application-specific business logic
    TrmUserTransaction transaction = new TrmUserTransaction();
    //Starts a transaction. This is where the transaction boundary begins.
```

```

    transaction.begin();
    // code that is specific to the application's business logic
    // withdraw operation on participant service 1
    isWithdrawSuccessful = withdraw();
    if(!isWithdrawSuccessful){
        transaction.rollback();
    }
    // deposit operation on participant service 2
    isDepositSuccessful = deposit();
    if(!isDepositSuccessful){
        transaction.rollback();
    }
    transaction.commit();
    //When you commit or rollback a transaction, the transaction boundary
ends.
    //Application-specific business logic
}

```

Sample Code for a JAX-RS Application that Defines Transaction Boundaries

Create and handle the `MicroTxUserTransactionService` object.

```

// Autowire the MicroTx transaction service
@Autowired
MicroTxUserTransactionService microTxtransaction;

transfer() {
    try {
        //Starts a transaction. This is where the transaction boundary begins.
        microTxtransaction.begin();
        // code that is specific to the application's business logic
        // withdraw operation on participant service 1
        isWithdrawSuccessful = withdraw();

        if(!isWithdrawSuccessful){
            microTxtransaction.rollback();
        }
        // deposit operation on participant service 2
        isDepositSuccessful = deposit();
        if(!isDepositSuccessful){
            microTxtransaction.rollback();
        }
        microTxtransaction.commit();
        //When you commit or rollback a transaction, the transaction boundary
ends.
        //Application-specific business logic
    } catch (Exception ex) {
        microTxtransaction.rollback();
        // Handle exception
    }
}

```

All code examples in this guide define the transaction boundary.

3.10 Develop Spring Boot Apps with XA

Use the MicroTx Distributed Transactions library with your Spring Boot applications.

You can use Spring REST or JAX-RS to develop your Spring Boot applications.

The procedure to integrate the MicroTx Distributed Transactions library files remains almost the same in both cases. When you develop Spring Boot applications based on JAX-RS, you must include different MicroTx Distributed Transactions Maven dependencies and you must register filters so that the MicroTx Distributed Transactions library can intercept HTTP calls and manage the transaction.

Topics

- [Configure Library Properties for Spring Boot Apps](#)
Provide configuration information for the MicroTx Distributed Transactions library properties for every Spring Boot application. The property values that you must provide would vary depending on whether the application is a participant or an initiator.
- [Configure Spring Boot App as Transaction Initiator](#)
A transaction initiator service initiates or starts a transaction. Based on your application's business logic, a transaction initiator service may only start the transaction or start the transaction and participate in the transaction as well.
- [Configure Spring Boot App as Transaction Participant](#)
Based on whether the resource manager is compliant with XA or not, set environment variables and implement different classes from the MicroTx Distributed Transactions library to configure your Spring Boot participant application.

3.10.1 Configure Library Properties for Spring Boot Apps

Provide configuration information for the MicroTx Distributed Transactions library properties for every Spring Boot application. The property values that you must provide would vary depending on whether the application is a participant or an initiator.

Provide property values for the MicroTx Distributed Transactions library in the `application.yaml` file.

- `spring.microtx.coordinator-url`: Enter the URL to access the MicroTx Distributed Transactions coordinator. See [Access MicroTx Distributed Transactions](#). You must enter this value for the transaction initiator application. You don't have to specify this value for the transaction participant applications.
- `spring.microtx.participant-url`: Enter the URL of your participant service. MicroTx Distributed Transactions uses the URL that you provide to connect to the participant service. Provide this value in the following format:

```
https://externalHostnameOfApp:externalPortOfApp/
```

```
https://externalHostnameOfApp:externalPortOfApp/
```

Where,

- `externalHostnameOfApp`: The external host name of your initiator or participant service. For example, `bookTicket-app`.

- *externalPortOfApp*: The port number over which you can access your initiator or participant service remotely. For example, 8081.

You must specify this value for the transaction participant applications.

For example, `https://bookTicket-app:8081`.

- `spring.microtx.propagation-active`: Set this to true when you want to trace the transaction from end-to-end. This propagates the trace headers for all incoming and outgoing requests.
- `spring.microtx.http-client-connection-pool-size`: Enter the number of connections to the MicroTx Distributed Transactions library to MicroTx Distributed Transactions. The default and minimum number of connections is 10. The maximum value is 20. You can change this value depending on the number of queries that your services run. Specify this value for both initiator and participant applications.
- `spring.microtx.xa-transaction-timeout`: Specify the maximum amount of time, in milliseconds, for which the transaction remains active. If a transaction is not committed or rolled back within the specified time period, the transaction is rolled back. The default value and minimum value is 60000. Specify this value only for the transaction initiator application. The transaction initiator service propagates the timeout value, that you have specified, to the participant services. If you specify this value for a participant application, it is ignored. See [Set Transaction Timeout](#).
- `spring.microtx.xa-resource-manager-id`: Specify a unique string value for each resource manager that you use in the XA transaction. The unique value that you provide as RMID is used by MicroTx Distributed Transactions to identify the resource manager. If more than one participant uses the same resource manager, then specify the same resource manager ID for the participants that share a resource manager. This value is not related to any properties of the data store. For example, 174A5FF2-D8B2-47B0-AF09-DA5AFECA2F71.
- `spring.microtx.xa-xa-support`: Set this to true when you use XA-compliant resources. Set this to false only for the single transaction participant service that uses a non-XA resource. The default value is true. When `xa-xa-support` is set to true, the values set for `xa-llr-support` and `xa-lrc-support` are ignored.
- `spring.microtx.xa-llr-support`: Set this to true to enable the Logging Last Resource (LLR) optimization. Set this value only for the transaction participant service that uses a non-XA resource as a resource manager. The default value is false. When `xa-llr-support` is set to true, the value set for `xa-lrc-support` is ignored. Only when you set this property value to true, you can also specify a value for the `xa-llr-delete-commit-record-interval` property. Specify the maximum amount of time, in milliseconds, for which the committed records are retained. The default value is 7,200,000 ms or 2 hours. Before performing a local commit, the transaction coordinator creates a commit record in the LLR branch. The commit records are deleted after the time period specified in the `xa-llr-delete-commit-record-interval` property. Specify this value only for the service that uses a non-XA resource as a resource manager.
- `spring.microtx.xa-lrc-support`: Set this to true to enable the Last Resource Commit (LRC) optimization. Set this value only for the transaction participant service that uses a non-XA resource as a resource manager. The default value is false.
- `spring.microtx.xa-promotable-active`: Set this to true to enable local transactions or to manage certain transactions locally without involving the coordinator. Specify this value only for a transaction initiator service that also participates in the transaction. The default value is false. If you set the value of `xa-promotable-active` as false, every transaction starts as a global transaction by default and it is associated with a GTRID. A local transaction is not initiated. See [About Global and Local Transactions](#).

- `spring.microtx.xa-rac-active`: Set this to true only for the transaction participant service that uses an Oracle Real Application Clusters (RAC) database as a resource manager. The default value is false.

For example,

```
spring:
  microtx:
    coordinator-url: http://tmm-app:9000/api/v1
    participant-url: https://bookTicket-app:8081
    propagation-active: true
    http-client-connection-pool-size: 60
    xa-transaction-timeout: 60000
    xa-xa-support: true
    xa-llr-support: false
    xa-lrc-support: false
    xa-llr-delete-commit-record-interval: 720000
    xa-resource-manager-id: 174A5FF2-D8B1-47B0-AF09-DA5AFECA2F61
    xa-promotable-active: false
    xa-rac-active: false
```

You can use the HTTP protocol if your application and MicroTx Distributed Transactions are in the same Kubernetes cluster, otherwise use the HTTPS protocol.

You can also provide these configuration values as environment variables. Note that if you specify values in both the `application.yaml` file as well as the environment variables, then the values set in the environment variables override the values in the properties file.

The following example provides sample values to configure the environment variables.

```
export SPRING_MICROTX_COORDINATOR_URL = http://tmm-app:9000/api/v1
export SPRING_MICROTX_PARTICIPANT_URL = http://bookTicket-app:8081
export SPRING_MICROTX_PROPAGATION_ACTIVE = true
export SPRING_MICROTX_HTTP_CLIENT_CONNECTION_POOL_SIZE = 15
export SPRING_MICROTX_XA_TRANSACTION_TIMEOUT = 60000
export SPRING_MICROTX_XA_XA_SUPPORT = true
export SPRING_MICROTX_XA_LLRL_SUPPORT = false
export SPRING_MICROTX_XA_LLRL_DELETE_COMMIT_RECORD_INTERVAL = 720000
export SPRING_MICROTX_XA_LRC_SUPPORT = false
export SPRING_MICROTX_XA_RESOURCE_MANAGER_ID = 174A5FF2-D8B1-47B0-AF09-DA5AFECA2F61
export SPRING_MICROTX_PROMOTABLE_ACTIVE = false
export SPRING_MICROTX_XA_RAC_ACTIVE = false
```

Note that the environment variables names are case-sensitive.

3.10.2 Configure Spring Boot App as Transaction Initiator

A transaction initiator service initiates or starts a transaction. Based on your application's business logic, a transaction initiator service may only start the transaction or start the transaction and participate in the transaction as well.

Before you begin, identify if your application only initiates the transaction or initiates and participates in the transaction. Configure your application accordingly as the requirements vary slightly for the two scenarios.

Let us consider two scenarios to understand if your application only initiates the transaction or participates in the transaction as well.

- Scenario 1: A banking teller application transfers an amount from one department to another. Here, the teller application only initiates the transaction and does not participate in it. Based on the business logic, the teller application calls different services to complete the transaction. A database instance may or may not be attached to the teller application.
- Scenario 2: A banking teller application transfers an amount from one department to another. For every transaction, the teller application charges 1% as commission. Here, the teller application initiates the transaction and participates in it. A database instance must be attached to the teller application to save the transaction information.

To configure your Spring Boot 3.x application as a transaction initiator using Spring REST or JAX-RS:

1. Specify property values for the MicroTx Distributed Transactions library. See [Configure Library Properties for Spring Boot Apps](#).
2. Include MicroTx Distributed Transactions library files as a maven dependency in the Spring Boot 3.x application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

-
- [Spring REST](#)
 - [Spring JAX-RS](#)

Spring REST

- For JDBC applications, use the microtx-spring-boot-starter file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the microtx-spring-boot-starter and microtx-spring-boot-starter-hibernate files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>

<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the `microtx-spring-boot-starter` and `microtx-spring-boot-starter-eclipselink` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>

<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

Spring JAX-RS

- For JDBC applications, use the `microtx-spring-boot-jaxrs-starter` file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the `microtx-spring-boot-jaxrs-starter` and `microtx-spring-boot-starter-hibernate` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>

<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the `microtx-spring-boot-jaxrs-starter` and `microtx-spring-boot-starter-eclipselink` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>

<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

3. Only for Spring JAX-RS applications, you must register the following filters so that the MicroTx Distributed Transactions library can intercept the JAX-RS calls and manage the transaction. You can skip this step for Spring REST applications.

```
import org.glassfish.jersey.server.ResourceConfig;
...
@Component
public class Configuration extends ResourceConfig {
    public Configuration() {
        ...
        register(TrmTransactionRequestFilter.class);
        register(TrmTransactionResponseFilter.class);
        register(JaxRsXACallbackResource.class);
    }
}
```

4. Complete the following steps if you want to use the `@Transactional` annotation. See [About @Transactional](#).

- a. Import the `org.springframework.transaction.annotation.Transactional` package.

```
import org.springframework.transaction.annotation.Transactional;
```

- b. Annotate the method which initiates the transaction with `@Transactional`. The following code sample demonstrates the usage of `@Transactional`.

```
import org.springframework.transaction.annotation.Transactional;
...
@RequestMapping(value = "", method = RequestMethod.POST)
@Transactional(propagation = Propagation.REQUIRED, timeout = 60)
public ResponseEntity<?> transfer(@RequestBody Transfer
transferDetails) {
    // code that is specific to the application's business logic
    // withdraw operation on participant service 1
    // deposit operation on participant service 2
}
```

For details about using `@Transactional` for Spring Boot apps, see <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.transaction.annotation.Transactional.html>.

- c. Ensure that the method returns a `ResponseEntity` object as shown in the example. This is required by the MicroTx Distributed Transactions library to handle the transaction and trigger the `ResponseBodyFilter` class. In MicroTx Distributed Transactions, the `com.oracle.microtx.springboot.interceptor.ResponseBodyFilter` class is declared with `@ControllerAdvice`.
- d. Skip to step 9 as you don't have to specify the transaction boundaries using `begin()`, `commit()`, and `rollback()` when you use `@Transactional`.
5. Import the `com.oracle.microtx.xa.rm.MicroTxUserTransactionService` package.

```
import com.oracle.microtx.xa.rm.MicroTxUserTransactionService;
```

6. Autowire the `MicroTxUserTransactionService` class before your application logic initiates or begins a transaction.

```
@Autowired
MicroTxUserTransactionService microTxUserTransactionService;
```

7. Optional. Specify the amount of time, in milliseconds, for which the transaction remains active. If a transaction is not committed or rolled back within the specified time period, the transaction is rolled back. This overrides the `spring.microtx.xa-transaction-timeout` property value in the `application.yaml` file. See [Set Transaction Timeout](#).

```
microTxUserTransaction.setTransactionTimeout(60000);
```

8. Ensure that the method returns a `ResponseEntity` object as shown in the example.

```
@RequestMapping(value = "", method = RequestMethod.POST)
public ResponseEntity<?> transfer(@RequestBody Transfer transferDetails)
{
    ...
}
```

This is required by the MicroTx Distributed Transactions library to handle the transaction and trigger the `ResponseBodyFilter` class. In MicroTx Distributed Transactions, the `com.oracle.microtx.springboot.interceptor.ResponseBodyFilter` class is declared with `@ControllerAdvice`.

9. Within the application code that initiates the transaction, demarcate the transaction boundaries, such as to begin, commit, or roll back the transactions. The following code sample demonstrates how to begin a new XA transaction.
 - If your application only initiates the transaction and does not participate in the transaction, add the following line to your application code to begin the XA transaction.

```
microTxUserTransactionService.begin();
... // Implement the business logic to begin a transaction.
```

- If your application initiates the transaction and participates in it, add the following line to your application code to begin the XA transaction.

```
microTxUserTransactionService.begin(true);
... // Implement the business logic to begin a transaction.
```

10. Based on your business logic, commit or rollback the transaction.

- To commit a transaction:

```
microTxUserTransactionService.commit();
```

- To rollback a transaction:

```
microTxUserTransactionService.rollback();
```

The following code sample demonstrates how you can demarcate the transaction boundaries using `begin()`, `commit()`, and `rollback()`.

```
@RequestMapping(value = "", method = RequestMethod.POST)
public ResponseEntity<?> transfer(@RequestBody Transfer transferDetails) {
    microTxUserTransactionService.begin();
    // app specific code
    try {
        microTxUserTransactionService.commit();
    } catch (Exception e) {
        microTxUserTransactionService.rollback();
    }
}
```

11. Use a REST template or OpenFeign to call the endpoints of the transaction participant services to perform the transaction. The transaction initiator service begins the transaction. To complete the transaction, an initiator service may have to call one or more participant services. While calling the participant services, use the object that you create.

-
- [RestTemplate](#)
 - [OpenFeign](#)

RestTemplate

Autowire the Spring Boot RestTemplate, provided by the MicroTx Distributed Transactions library as shown in the following sample code.

```
@Autowired
@Qualifier("MicroTxXaRestTemplate")
RestTemplate restTemplate;
```

OpenFeign

Configure an interceptor, provided by the MicroTx Distributed Transactions library, in one of the following ways. This is required if an outgoing request is a part of a transaction or if you want to intercept an outgoing request.

- Configure an interceptor in the configuration property of the `@FeignClient` annotation as shown in the following example.

```
@FeignClient(value = "<feign1.name>", url = "<feign1.url>",
configuration = MicroTxXaFeignInterceptorConfig.class)
public interface <interface_name> {
    // ...
}
```

Where, you replace `<feign1.name>` and `<feign1.url>` with values specific to your environment. Note down the name of the interface as you will use it later in your application code.

- Configure interceptors by specifying property values as shown in the following example. Provide these values in the same file that contains values for the MicroTx Distributed Transactions client library properties.

```
spring.cloud.openfeign.client.config.<feign1_name>.request-  
interceptors[0]=  
com.oracle.microtx.springboot.xa.clientinterceptor.MicroTxXaFeignInterce  
ptorConfig
```

Where, *<feign1_name>* is the data for the value property of the `@FeignClient` annotation.

12. Save your changes.

Source code of a sample Spring REST transaction initiator application which uses the MicroTx Distributed Transactions library and XA is available in the [teller-spring](#) folder which is located in the microtx-samples GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application. This provides an example of how you can use the MicroTx Distributed Transactions Spring REST libraries with the business logic of your Spring REST initiator application. This sample application is called Teller. It initiates a transaction between two departments. It calls Dept A to withdraw an amount and it calls Dept B to deposit the amount.

If the initiator service also participates in the transaction in addition to initiating the transaction, you must make additional configurations for the application to participate in the transaction and communicate with the resource manager. See [Configure Spring Boot App as Transaction Participant](#). The [teller-as-participant-spring](#) folder which is located in the microtx-samples GitHub repository provides the source code of a Spring REST transaction initiator application which also participates in the transaction.

3.10.3 Configure Spring Boot App as Transaction Participant

Based on whether the resource manager is compliant with XA or not, set environment variables and implement different classes from the MicroTx Distributed Transactions library to configure your Spring Boot participant application.

① Note

Ensure that the response body is not null for all the REST APIs that participate in the transaction.

- [Configure Spring Boot Apps with an XA-Compliant Resource Manager](#)
Use the information provided in this section to configure your Spring Boot participant applications when you use an XA-compliant resource manager.
- [Configure Spring Boot App with Multiple XA-Compliant Resource Managers](#)
Use the information provided in this section to configure your Spring Boot participant applications when you use multiple XA-compliant resource managers.
- [Configure Spring Boot App with a Non-XA JDBC Resource](#)
Use the information provided in this section to configure your Spring Boot participant applications when you use a JDBC resource that does not support XA.

- [Configure Spring Boot App with a Non-XA and Non-JDBC Resource](#)
Use the information provided in this section to configure your Spring Boot participant applications when you use a resource that does not support XA and JDBC.
- [Configure JPA-Based Spring Boot Apps with XA-Compliant Resource Manager](#)
Use the information provided in this section to configure Hibernate or EclipseLink as the JPA provider for your Helidon or Spring Boot application which participates in an XA transaction when you use an XA-compliant resource manager.

3.10.3.1 Configure Spring Boot Apps with an XA-Compliant Resource Manager

Use the information provided in this section to configure your Spring Boot participant applications when you use an XA-compliant resource manager.

1. Configure property values for the MicroTx Distributed Transactions client library.

The following example provides sample values for the properties. Provide the values based on your environment.

```
spring:
  microtx:
    participant-url: https://bookTicket-app:8081
    propagation-active: true
    http-client-connection-pool-size: 15
    xa-transaction-timeout: 60000
    xa-resource-manager-id: 174A5FF2-D8B1-47B0-AF09-DA5AFECA2F61
    xa-xa-support: true
```

Ensure that xa-xa-support is set to true.

For details about each property and other optional properties, see [Configure Library Properties for Spring Boot Apps](#).

2. Include MicroTx Distributed Transactions library files as a maven dependency in the Spring Boot 3.x application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

-
- [Spring REST](#)
 - [Spring JAX-RS](#)

Spring REST

- For JDBC applications, use the microtx-spring-boot-starter file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the `microtx-spring-boot-starter` and `microtx-spring-boot-starter-hibernate` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>

<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the `microtx-spring-boot-starter` and `microtx-spring-boot-starter-eclipselink` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>

<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

Spring JAX-RS

- For JDBC applications, use the `microtx-spring-boot-jaxrs-starter` file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the `microtx-spring-boot-jaxrs-starter` and `microtx-spring-boot-starter-hibernate` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>

<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the `microtx-spring-boot-jaxrs-starter` and `microtx-spring-boot-starter-eclipselink` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>

<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

3. Only for Spring JAX-RS applications, you must register the following filters so that the MicroTx Distributed Transactions library can intercept the JAX-RS calls and manage the transaction. You can skip this step for Spring REST applications.

```
import org.glassfish.jersey.server.ResourceConfig;
...
@Component
public class Configuration extends ResourceConfig {
    public Configuration() {
        ...
        register(TrmTransactionRequestFilter.class);
        register(TrmTransactionResponseFilter.class);
        register(JaxRsXACallbackResource.class);
    }
}
```

4. Import the `com.oracle.microtx.common.MicroTxConfig` package.

```
import com.oracle.microtx.common.MicroTxConfig;
```

5. Initialize an `XADatasource` object.

The MicroTx Distributed Transactions client library needs to access an `XADatasource` object. It uses this object to create `XAConnection` and `XAResource` objects to connect with a resource manager or database server. The following code describes how you can define the `XADatasource` object at the beginning of the application code when you create the connection object.

```
class oracle.tmm.jta.MicroTxConfig
static void initXaDataSource(XADatasource xaDs)
```

For more information about `XADatasource`, see <https://docs.oracle.com/javase/8/docs/api/javax/sql/XADatasource.html>.

6. In the transaction participant function or block, specify the XADataSource object which is used by the MicroTx Distributed Transactions client library. Provide the credentials and other details to connect to the resource manager.

```
//Example for a participant using an Oracle Database:
OracleXADataSource dataSource = new
oracle.jdbc.xa.client.OracleXADataSource();
dataSource.setURL(url); //database connection string
dataSource.setUser(user); //username to access database
dataSource.setPassword(password); //password to access database
MicroTxConfig.initXADataSource((XADataSource)dataSource);
```

It is the responsibility of the application developer to ensure that an XA-compliant JDBC driver and required parameters are set up while allocating XADataSource.

The MicroTx Distributed Transactions client library uses the XADataSource object to create database connections.

7. In the transaction participant function or block, add the following line of code only once after you have initialized the XADataSource object.

```
oracle.tmm.jta.MicroTxConfig.initXADataSource((XADataSource)xads);
```

XADataSource is an interface defined in JTA whose implementation is provided by the JDBC driver.

The MicroTx Distributed Transactions client library uses this object to connect to database to start XA transactions and perform various operations such as prepare, commit, and rollback. The MicroTx Distributed Transactions library also provides a SQL connection object to the application code to execute DML using dependency injection.

8. Insert the following line in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions client library. The following code in the participant application injects the connection object that is created by the MicroTx Distributed Transactions client library.

```
@Autowired
@Qualifier("microTxSqlConnection")
@Lazy
private Connection connection;
```

9. Insert the following lines in the code of the participant service so that the service uses the injected connection object whenever the participant service performs a DML operation.

```
Statement stmt1 = connection.createStatement();
stmt1.execute(query);
stmt1.close();
```

Where, connection is the name of the Connection object that you have injected in the previous step.

Insert these lines of code for every DML operation that your participant service performs. Create a new statement object, such as stmt1 or stmt2 for every DML operation, but use the same connection object that is created by the MicroTx Distributed Transactions client library.

10. Save the changes.

If there are multiple Spring Boot services complete these steps for all the transaction participant services.

Source code of a sample Spring REST transaction participant application which uses the MicroTx Distributed Transactions library and XA transaction protocol is available in the [department-spring](#) folder which is located in the microtx-samples GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.10.3.2 Configure Spring Boot App with Multiple XA-Compliant Resource Managers

Use the information provided in this section to configure your Spring Boot participant applications when you use multiple XA-compliant resource managers.

Your application can connect to multiple XA-compliant resource managers. Additionally, your application can connect to a single non-XA resource.

1. Include MicroTx Distributed Transactions library files as a maven dependency in the Spring Boot 3.x application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

-
- [Spring REST](#)
 - [Spring JAX-RS](#)

Spring REST

- For JDBC applications, use the microtx-spring-boot-starter file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the microtx-spring-boot-starter and microtx-spring-boot-starter-hibernate files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the microtx-spring-boot-starter and microtx-spring-boot-starter-eclipselink files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
```

```
<artifactId>microtx-spring-boot-starter</artifactId>
<version>26.1</version>
</dependency>

<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

Spring JAX-RS

- For JDBC applications, use the microtx-spring-boot-jaxrs-starter file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the microtx-spring-boot-jaxrs-starter and microtx-spring-boot-starter-hibernate files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the microtx-spring-boot-jaxrs-starter and microtx-spring-boot-starter-eclipselink files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

- Only for Spring JAX-RS applications, you must register the following filters so that the MicroTx Distributed Transactions library can intercept the JAX-RS calls and manage the transaction. You can skip this step for Spring REST applications.

```
import org.glassfish.jersey.server.ResourceConfig;
...
@Component
public class Configuration extends ResourceConfig {
    public Configuration() {
        ...
        register(TrmTransactionRequestFilter.class);
        register(TrmTransactionResponseFilter.class);
        register(JaxRsXACallbackResource.class);
    }
}
```

- Import the `com.oracle.microtx.common.MicroTxConfig` package.

```
import com.oracle.microtx.common.MicroTxConfig;
```

- Create a `DataSourceInfo` object for each resource manager. Ensure that you provide the data source names and resource manager IDs that you have specified in the your application's YAML file.

Sample Command

```
DataSourceInfo departmentDataSourceInfo = new
DataSourceInfo("ORCL1-8976-9776-9873");
departmentDataSourceInfo.setDataSourceName(departmentDataSource);

DataSourceInfo creditDataSourceInfo = new
DataSourceInfo("ORCL2-2134-5668-8672");
creditDataSourceInfo.setDataSourceName(creditDataSource);
```

Where,

- `departmentDataSource` and `creditDataSource` are names of the XA data source that you have provided in your application's YAML file.
- `ORCL1-8976-9776-9873` and `ORCL2-2134-5668-8672` are the resource manager IDs that you have provided respectively for `departmentDataSource` and `creditDataSource` in your application's YAML file.

Later, you will use the `@Inject` annotation to ensure that your application uses these data sources.

- Enter only *one* of the following MicroTx Distributed Transactions client library properties for each `DataSourceInfo` object that you have created. The following example provides property value for the `creditDataSource` object. Similarly, you can provide property values for other resource managers. If you don't provide any value, by default the resource is considered to be XA-compliant.
- For XA-compliant resources, enter:

```
creditDataSource.setXaSupport();
```

- For Oracle RAC database, enter:

```
creditDataSource.setRAC(true);
```

- For non-XA resources that use LLR optimization, enter:

```
creditDataSource.setLLRSupport();
```

- For non-XA resources that use LRC optimization, enter:

```
creditDataSource.setLRCSupport();
```

6. Initialize an XADataSource object. If you are using multiple resource managers with your application, initialize the XADataSource object in the following way for every XA-compliant resource manager.

The MicroTx Distributed Transactions client library needs to access an XADataSource object. It uses this object to create XAConnection and XAResource objects to connect with a resource manager or database server. The following code describes how you can define the XADataSource object at the beginning of the application code when you create the connection object.

```
class oracle.tmm.jta.MicroTxConfig
static void initXADataSource(XADataSource dataSource, DataSourceInfo
creditDataSource)
```

Where, creditDataSource is the DataSourceInfo object that you have previously created.

Repeat this step for every resource manager.

For more information about XADataSource, see <https://docs.oracle.com/javase/8/docs/api/javax/sql/XADataSource.html>.

7. In the transaction participant function or block, specify the XADataSource object which is used by the MicroTx Distributed Transactions client library. Provide the credentials and other details to connect to the resource manager.

```
//Example for a participant using an Oracle Database:
OracleXADataSource dataSource = new
oracle.jdbc.xa.client.OracleXADataSource();
dataSource.setURL(url); //database connection string
dataSource.setUser(user); //username to access database
dataSource.setPassword(password); //password to access database
```

It is the responsibility of the application developer to ensure that an XA-compliant JDBC driver and required parameters are set up while allocating XADataSource.

The MicroTx Distributed Transactions client library uses the XADataSource object to create database connections.

Repeat this step for every resource manager.

8. In the transaction participant function or block, add the following line of code only once after you have initialized the XADataSource object.

```
oracle.tmm.jta.MicroTxConfig.initXADataSource(dataSource, creditDataSource)
```

Where, *creditDataSource* is the *DataSourceInfo* object that you have previously created.

The MicroTx Distributed Transactions client library uses this object to connect to database to start XA transactions and perform various operations such as prepare, commit, and rollback. The MicroTx Distributed Transactions library also provides a SQL connection object to the application code to execute DML using dependency injection.

Repeat this step for every resource manager.

9. Insert the following line in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions client library. The following code in the participant application injects the connection object that is created by the MicroTx Distributed Transactions client library.

If you are using multiple resource managers with your application, inject a connection object in the following way for every XA-compliant resource manager.

The following example describes how you can add two connections, one for each XA-compliant resource manager. When you inject the connection, call *microTxSqlConnection* followed by the name of your data source.

```
//Initialize the application context
private ApplicationContext applicationContext;
//Connection for XA-compliant resource manager 1
private Connection connection;
//Connection for XA-compliant resource manager 2
private Connection creditConnection;

@Autowired
public AccountService(ApplicationContext applicationContext) {
    this.applicationContext = applicationContext;
    try {
        // Inject the connections for XA-compliant resource manager 1.
        // Specify the name of the data source associated with XA-
        compliant resource manager 1.
        connection = (Connection)
        applicationContext.getBean("microTxSqlConnection", "departmentDataSource");
        // Inject the connections for XA-compliant resource manager 2.
        // Specify the name of the data source associated with XA-
        compliant resource manager 2.
        creditConnection = (Connection)
        applicationContext.getBean("microTxSqlConnection", "creditDataSource");
    } catch (ClassCastException ex) {
        LOG.info(ex.getMessage());
    }
}
```

Where, *creditDataSource* is the value that you have provided for the *dataSourceName* string in the *DataSourceInfo* class of the *oracle.tmm.jta.common.DataSourceInfo* package.

Repeat this step for every resource manager.

10. Insert the following lines in the code of the participant service so that the service uses the injected connection object whenever the participant service performs a DML operation.

```
Statement stmt1 = creditConnection.createStatement();  
stmt1.execute(query);  
stmt1.close();
```

Where, `creditConnection` is the name of the `Connection` object that you have injected in the previous step.

Insert these lines of code for every DML operation that your participant service performs. Create a new statement object, such as `stmt1` or `stmt2` for every DML operation, but use the same `creditConnection` object that is created by the MicroTx Distributed Transactions client library.

Repeat this step for every resource manager.

11. Save the changes.

If there are multiple Spring Boot transaction participant services complete these steps for all the participant services.

Source code of a sample Spring REST transaction participant application which uses the MicroTx Distributed Transactions library and XA transaction protocol is available in the [department-spring-multiplerm](#) folder which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.10.3.3 Configure Spring Boot App with a Non-XA JDBC Resource

Use the information provided in this section to configure your Spring Boot participant applications when you use a JDBC resource that does not support XA.

Your application can connect to multiple XA-compliant resource managers. Additionally, your application can connect to a single non-XA resource.

1. When you use a single resource manager, provide values for all the MicroTx Distributed Transactions client library properties in a single file, such as `application.yaml` file. Skip this step if you are using multiple resource managers for your application.

Ensure that `xa-xa-support` is set to `false` and `xa-llr-support` or `xa-lrc-support` is set to `true`.

- To enable the Logging Last Resource (LLR) optimization, set the following values for the following properties.

```
spring:  
  microtx:  
    xa-xa-support = false  
    xa-llr-support = true  
    xa-lrc-support = false  
    http-client-connection-pool-size = 15  
    participant-url = https://bookHotel-app:8081  
    propagation-active = true  
    xa-transaction-timeout = 60000
```

- To enable the Last Resource Commit (LRC) optimization, set the values for the following properties.

```
spring:
  microtx:
    xa-xa-support = false
    xa-llr-support = false
    xa-lrc-support = true
    http-client-connection-pool-size = 15
    participant-url = https://bookHotel-app:8081
    propagation-active = true
    xa-transaction-timeout = 60000
```

For details about each property and other optional properties, see [Configure Library Properties for Spring Boot Apps](#).

2. If you are using a multiple resource managers with your application, complete the following steps to configure property values for the MicroTx Distributed Transactions client library. Skip this step if you are using a single resource manager.
 - a. Create a `DataSourceInfo` object for each resource manager. Ensure that you provide the data source names and resource manager IDs that you have the specified in the your application's YAML file.

Sample Command

```
DataSourceInfo departmentDataSourceInfo = new
DataSourceInfo("ORCL1-8976-9776-9873");
departmentDataSourceInfo.setDataSourceName(departmentDataSource);

DataSourceInfo creditDataSourceInfo = new
DataSourceInfo("ORCL2-2134-5668-8672");
creditDataSourceInfo.setDataSourceName(creditDataSource);
```

Where,

- `departmentDataSource` and `creditDataSource` are names of the XA data source that you have provided in your application's YAML file.
- `ORCL1-8976-9776-9873` and `ORCL2-2134-5668-8672` are the resource manager IDs that you have provided respectively for `departmentDataSource` and `creditDataSource` in your application's YAML file.

Later, you will use the `@Inject` annotation to ensure that your application uses these data sources.

- b. Enter only *one* of the following MicroTx Distributed Transactions client library properties for each `DataSourceInfo` object that you have created. The following example provides property value for the `creditDataSource` object. Similarly, you can provide property values for other resource managers. If you don't provide any value, by default the resource is considered to be XA-compliant.
 - For XA-compliant resources, enter `creditDataSource.setXaSupport();`.
 - For non-XA resources that use LLR optimization, enter `creditDataSource.setLLRSupport();`.
 - For non-XA resources that use LRC optimization, enter `creditDataSource.setLRCSupport();`.

3. Include MicroTx Distributed Transactions library files as a maven dependency in the Spring Boot 3.x application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

-
- [Spring REST](#)
 - [Spring JAX-RS](#)

Spring REST

- For JDBC applications, use the microtx-spring-boot-starter file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the microtx-spring-boot-starter and microtx-spring-boot-starter-hibernate files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the microtx-spring-boot-starter and microtx-spring-boot-starter-eclipselink files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

Spring JAX-RS

- For JDBC applications, use the `microtx-spring-boot-jaxrs-starter` file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the `microtx-spring-boot-jaxrs-starter` and `microtx-spring-boot-starter-hibernate` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the `microtx-spring-boot-jaxrs-starter` and `microtx-spring-boot-starter-eclipselink` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

- Only for Spring JAX-RS applications, you must register the following filters so that the MicroTx Distributed Transactions library can intercept the JAX-RS calls and manage the transaction. You can skip this step for Spring REST applications.

```
import org.glassfish.jersey.server.ResourceConfig;
...
@Component
public class Configuration extends ResourceConfig {
    public Configuration() {
        ...
        register(TrmTransactionRequestFilter.class);
        register(TrmTransactionResponseFilter.class);
    }
}
```

```
        register(JaxRsXACallbackResource.class);
    }
}
```

5. Enable session affinity. See [Enable Session Affinity](#).
6. Import the `com.oracle.microtx.common.MicroTxNonXAConfig` package.

```
import com.oracle.microtx.common.MicroTxNonXAConfig;
```

7. Initialize a Datasource object.

The MicroTx Distributed Transactions library needs to access a data source object. It uses the data source object to create `java.sql.Connection` objects to connect with a resource manager. The following code describes how you can define a data source object.

You must provide this code at the start of the application, so that the `initNonXaDataSource` method is called immediately after the server starts and before any other requests are served.

- If you are using a single resource manager with your application, initialize a data source in the following way.

```
class oracle.tmm.jta.MicroTxNonXAConfig
static void initNonXaDataSource(DataSource NonXaDs)
```

- If you are using multiple resource managers with your application, initialize the data source object in the following way for the Non-XA JDBC resource. A participant service can connect to multiple XA-compliant resource managers, but only one non-XA resource is supported in a transaction.

```
class oracle.tmm.jta.MicroTxNonXAConfig
static void initNonXaDataSource(DataSource departmentDataSource,
DataSourceInfo departmentDataSourceInfo)
```

Where, *dataSourceInfo* is the object that you have created in the step 2.

8. In the transaction participant function or block, specify the `DataSource` object which is used by the MicroTx Distributed Transactions library. Provide the credentials and database driver details to connect to the resource manager. The following example shows the details that you must provide when you use MySQL database as an LLR. Similarly, you can provide credentials and database driver information for other databases.

```
//Example for a participant using a MySQL database as resource manager
this.dataSource = PoolDataSourceFactory.getPoolDataSource();
this.dataSource.setURL(url); //Database connection string
this.dataSource.setUser(user); //User name to access the database
this.dataSource.setPassword(password); //Password to access the database
//Database driver information for the MySQL database.
//Provide the JDBC driver information that is specific to your database.
this.dataSource.setConnectionFactoryClassName("com.mysql.cj.jdbc.MysqlDataS
ource");
this.dataSource.setMaxPoolSize(15);
```

It is the application developer's responsibility to ensure that a database-specific JDBC driver and required parameters are set up while allocating `DataSource`.

MicroTx Distributed Transactions library uses the DataSource object to create database connections.

9. In the transaction participant function or block, add the following line of code only once after you have initialized the Datasource object. The MicroTx Distributed Transactions library uses this object to start a database transaction. The MicroTx Distributed Transactions library also provides a SQL connection object to the application code to execute DML using dependency injection.

```
oracle.tmm.jta.MicroTxNonXAConfig.initNonXaDataSource((DataSource)
NonXaDs);
```

Where, Datasource is an interface defined in JTA whose implementation is provided by the JDBC driver.

10. Insert the following line in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions library. The following code in the participant application injects the connection object that is created by the MicroTx Distributed Transactions library.

```
@Autowired
@Qualifier("microTxNonXASqlConnection")
@Lazy
private Connection connection;
```

11. Insert code in the participant service so that the service uses the injected connection object whenever the participant service performs a DML operation. You can create code to use the injected connection object based on your business scenario. Here's an example code snippet.

```
Statement stmt1 = connection.createStatement();
stmt1.execute(query);
stmt1.close();
```

Insert these lines of code for every DML operation that your participant service performs. Create a new statement object, such as stmt1 or stmt2 for every DML operation, but use the same connection object that's created by the MicroTx Distributed Transactions library.

12. Save the changes.

Source code of a sample Spring REST transaction participant application which uses the MicroTx Distributed Transactions library and XA transaction protocol is available in the [department-spring-nonxa-ds](#) folder which is located in the microtx-samples GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.10.3.4 Configure Spring Boot App with a Non-XA and Non-JDBC Resource

Use the information provided in this section to configure your Spring Boot participant applications when you use a resource that does not support XA and JDBC.

Your application can connect to multiple XA-compliant resource managers. However, only a single non-XA resource can participate in a transaction.

1. Before you begin, ensure that you have configured property values for the MicroTx Distributed Transactions library.

Ensure that `xa-xa-support` is set to `false` and `xa-llr-support` or `xa-lrc-support` is set to `true`.

- To enable the Logging Last Resource (LLR) optimization, set the following values for the following properties.

```
spring:
  microtx:
    xa-xa-support = false
    xa-llr-support = true
    xa-lrc-support = false
```

- To enable the Last Resource Commit (LRC) optimization, set values for the following properties.

```
spring:
  microtx:
    xa-xa-support = false
    xa-llr-support = false
    xa-lrc-support = true
```

For details about each property and other optional properties, see [Configure Library Properties for Spring Boot Apps](#).

2. Include MicroTx Distributed Transactions library files as a maven dependency in the Spring Boot 3.x application's `pom.xml` file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

-
- [Spring REST](#)
 - [Spring JAX-RS](#)

Spring REST

- For JDBC applications, use the `microtx-spring-boot-starter` file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the `microtx-spring-boot-starter` and `microtx-spring-boot-starter-hibernate` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
```

```

        <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
        <version>26.1</version>
    </dependency>

```

- When EclipseLink is the JPA provider, use the microtx-spring-boot-starter and microtx-spring-boot-starter-eclipselink files.

```

<dependency>
    <groupId>com.oracle.microtx</groupId>
    <artifactId>microtx-spring-boot-starter</artifactId>
    <version>26.1</version>
</dependency>

```

```

<dependency>
    <groupId>com.oracle.microtx</groupId>
    <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
    <version>26.1</version>
</dependency>

```

Spring JAX-RS

- For JDBC applications, use the microtx-spring-boot-jaxrs-starter file.

```

<dependency>
    <groupId>com.oracle.microtx</groupId>
    <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
    <version>26.1</version>
</dependency>

```

- When Hibernate is the JPA provider, use the microtx-spring-boot-jaxrs-starter and microtx-spring-boot-starter-hibernate files.

```

<dependency>
    <groupId>com.oracle.microtx</groupId>
    <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
    <version>26.1</version>
</dependency>

```

```

<dependency>
    <groupId>com.oracle.microtx</groupId>
    <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
    <version>26.1</version>
</dependency>

```

- When EclipseLink is the JPA provider, use the microtx-spring-boot-jaxrs-starter and microtx-spring-boot-starter-eclipselink files.

```

<dependency>
    <groupId>com.oracle.microtx</groupId>
    <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
    <version>26.1</version>
</dependency>

```

```

<dependency>
    <groupId>com.oracle.microtx</groupId>

```

```

        <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
        <version>26.1</version>
    </dependency>

```

3. Only for Spring JAX-RS applications, you must register the following filters so that the MicroTx Distributed Transactions library can intercept the JAX-RS calls and manage the transaction. You can skip this step for Spring REST applications.

```

import org.glassfish.jersey.server.ResourceConfig;
...
@Component
public class Configuration extends ResourceConfig {
    public Configuration() {
        ...
        register(TrmTransactionRequestFilter.class);
        register(TrmTransactionResponseFilter.class);
        register(JaxRsXACallbackResource.class);
    }
}

```

4. Enable session affinity. See [Enable Session Affinity](#).
5. Implement the NonXAResource interface.

```

public class MongoDbNonXAResource implements NonXAResource {
    // Provide application-specific code for all the methods in the
    NonXAResource interface.
}

```

For information about the NonXAResource interface, see [Transaction Manager for Microservices Java API Reference](#).

If you have enabled the LRC optimization, you don't have to implement the `recover()` method in the NonXAResource interface as the `commit()` method returns NULL for `commitRecord` in LRC.

6. After implementing the NonXAResource interface, import the MicroTx Distributed Transactions library files, and then produce a non-XA resource. Annotate the non-XA resource that you create with `@NonXa` annotation. The MicroTx Distributed Transactions library consumes the object that you annotate.

The following example shows a sample implementation for a MongoDB resource. Create code for your application based on your business requirements. In this example, the `NonXaResourceFactory` class supplies the NonXAResource. It produces a non-XA resource, and then the MicroTx Distributed Transactions library consumes the non-XA resource.

```

package com.oracle.mtm.sample.nonxa;

import oracle.tmm.jta.nonxa.NonXAResource;
import oracle.tmm.jta.nonxa.NonXa;

import javax.enterprise.inject.Produces;
import javax.inject.Inject;
import javax.ws.rs.ext.Provider;
import java.util.function.Supplier;

```

```

@Provider
public class NonXAResourceFactory implements Supplier<NonXAResource> {

    @Autowired(required = false)
    @Qualifier("NonXA")
    private NonXAResource trmNonXAInstance;

    @Produces
    @NonXa
    public NonXAResource getNonXAResource() {
        return nonXAResource;
    }

    @Override
    public NonXAResource get() {
        return getNonXAResource();
    }
}

```

7. Save the changes.

Source code of a sample Spring REST transaction participant application which uses the MicroTx Distributed Transactions library and XA transaction protocol is available in the [department-spring-nonxa-mongo](#) folder which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.10.3.5 Configure JPA-Based Spring Boot Apps with XA-Compliant Resource Manager

Use the information provided in this section to configure Hibernate or EclipseLink as the JPA provider for your Helidon or Spring Boot application which participates in an XA transaction when you use an XA-compliant resource manager.

Your application can connect to multiple XA-compliant resource managers. If you are using multiple XA-compliant resource managers for your application, complete the following steps for each resource manager.

1. Configure property values for the MicroTx Distributed Transactions client library.

The following example provides sample values for the properties. Provide the values based on your environment.

```

spring:
  microtx:
    participant-url: https://bookTicket-app:8081
    propagation-active: true
    http-client-connection-pool-size: 15
    xa-transaction-timeout: 60000
    xa-resource-manager-id: 174A5FF2-D8B1-47B0-AF09-DA5AFECA2F61
    xa-xa-support: true

```

Ensure that `xa-xa-support` is set to `true`.

For details about each property and other optional properties, see [Configure Library Properties for Spring Boot Apps](#).

2. Include MicroTx Distributed Transactions library files as a maven dependency in the Spring Boot 3.x application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

-
- [Spring REST](#)
 - [Spring JAX-RS](#)

Spring REST

- For JDBC applications, use the microtx-spring-boot-starter file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the microtx-spring-boot-starter and microtx-spring-boot-starter-hibernate files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the microtx-spring-boot-starter and microtx-spring-boot-starter-eclipselink files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

Spring JAX-RS

- For JDBC applications, use the `microtx-spring-boot-jaxrs-starter` file.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

- When Hibernate is the JPA provider, use the `microtx-spring-boot-jaxrs-starter` and `microtx-spring-boot-starter-hibernate` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
  <version>26.1</version>
</dependency>
```

- When EclipseLink is the JPA provider, use the `microtx-spring-boot-jaxrs-starter` and `microtx-spring-boot-starter-eclipselink` files.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-jaxrs-starter</artifactId>
  <version>26.1</version>
</dependency>
```

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
  <version>26.1</version>
</dependency>
```

- Only for Spring JAX-RS applications, you must register the following filters so that the MicroTx Distributed Transactions library can intercept the JAX-RS calls and manage the transaction. You can skip this step for Spring REST applications.

```
import org.glassfish.jersey.server.ResourceConfig;
...
@Component
public class Configuration extends ResourceConfig {
    public Configuration() {
        ...
        register(TrmTransactionRequestFilter.class);
        register(TrmTransactionResponseFilter.class);
    }
}
```

```
        register(JaxRsXACallbackResource.class);
    }
}
```

4. Create a .java file in the folder that contains your application code to initialize an XADataSourceConfig object. The XADataSourceConfig class contains methods to create custom data source and entity manager factory objects.

The custom data source object contains details to connect with the resource manager. It is the responsibility of the application developer to ensure that an XA-compliant JDBC driver and required parameters are set up while creating a custom data source object.

- The following example code for Hibernate apps using Spring Boot shows how you can initialize the library in within the XADataSourceConfig class, create a custom data source named departmentDataSource, and create an entity manager factory object named emf. You can create a similar code for your application.

Ensure that you specify

com.oracle.microtx.jpa.HibernateXADataSourceConnectionProvider as the property value for hibernate.connection.provider_class.

```
package com.oracle.mtm.sample;

import com.oracle.microtx.common.MicroTxConfig;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolXADataSource;
import org.hibernate.jpa.HibernatePersistenceProvider;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import javax.sql.XADataSource;
import java.sql.SQLException;
import java.util.Properties;

@Configuration
@ComponentScan("com.oracle")
public class XADataSourceConfig {
    @Value("${departmentDataSource.url}")
    private String url;
    @Value("${departmentDataSource.user}")
    private String username;
    @Value("${departmentDataSource.password}")
    private String password;
    @Value("${departmentDataSource.oracleucp.min-pool-size}")
    private String minPoolSize;
    @Value("${departmentDataSource.oracleucp.initial-pool-size:10}")
    private String initialPoolSize;

    @Value("${departmentDataSource.oracleucp.max-pool-size}")
    private String maxPoolSize;
```

```

@Value("${departmentDataSource.oracleucp.data-source-name}")
private String dataSourceName;

@Value("${departmentDataSource.oracleucp.connection-pool-name}")
private String connectionPoolName;

@Value("${departmentDataSource.oracleucp.connection-factory-class-
name:oracle.jdbc.xa.client.OracleXADataSource}")
private String connectionFactoryClassName;

@Value("${spring.microtx.xa-resource-manager-id}")
private String resourceId;

@Bean(name = "ucpXADataSource")
@Primary
public DataSource getDataSource() {
    DataSource pds = null;
    try {
        pds = PoolDataSourceFactory.getPoolXADataSource();

        ((PoolXADataSource)
pds).setConnectionFactoryClassName(connectionFactoryClassName);
        ((PoolXADataSource) pds).setURL(url);
        ((PoolXADataSource) pds).setUser(username);
        ((PoolXADataSource) pds).setPassword(password);
        ((PoolXADataSource)
pds).setMinPoolSize(Integer.valueOf(minPoolSize));
        ((PoolXADataSource)
pds).setInitialPoolSize(Integer.valueOf(initialPoolSize));
        ((PoolXADataSource)
pds).setMaxPoolSize(Integer.valueOf(maxPoolSize));

        ((PoolXADataSource) pds).setDataSourceName(dataSourceName);
        ((PoolXADataSource)
pds).setConnectionPoolName(connectionPoolName);

        System.out.println("XADataSourceConfig: XADataSource
created");
    } catch (SQLException ex) {
        System.err.println("Error connecting to the database: " +
ex.getMessage());
    }
    return pds;
}

@Bean(name = "entityManagerFactory")
public EntityManagerFactory createEntityManagerFactory() throws
SQLException {
    LocalContainerEntityManagerFactoryBean entityManagerFactoryBean
= new LocalContainerEntityManagerFactoryBean();

    entityManagerFactoryBean.setDataSource(getDataSource());
    entityManagerFactoryBean.setPackagesToScan(new String[]
{ "com.oracle.mtm.sample.entity" });
}

```

```

        entityManagerFactoryBean.setJpaVendorAdapter(new
HibernateJpaVendorAdapter());

entityManagerFactoryBean.setPersistenceProviderClass(HibernatePersistenc
eProvider.class);
        entityManagerFactoryBean.setPersistenceUnitName("mydeptxads");
        Properties properties = new Properties();
        properties.setProperty( "javax.persistence.transactionType",
"RESOURCE_LOCAL"); // change this to resource_local
        properties.put("hibernate.show_sql", "true");
        properties.put("hibernate.dialect",
"org.hibernate.dialect.Oracle12cDialect");
        properties.put("hibernate.format_sql", "true");
        properties.put("hbm2ddl.auto", "validate");
        properties.put("hibernate.connection.provider_class",
"com.oracle.microtx.jpa.HibernateXADataSourceConnectionProvider");
        entityManagerFactoryBean.setJpaProperties(properties);
        entityManagerFactoryBean.afterPropertiesSet();
        EntityManagerFactory emf = (EntityManagerFactory)
entityManagerFactoryBean.getObject();
        System.out.println("entityManagerFactory = " + emf);
        MicroTxConfig.initEntityManagerFactory(emf,
resourceManagerId); // Initialize TMM Library
        return emf;
    }
}

```

- The following example code for EclipseLink apps using Spring Boot shows how you can initialize the library in within the PoolXADataSource class, create a custom data source named departmentDataSource, and create an entity manager factory object named emf. You can create a similar code for your application.

Ensure that you specify values for the PersistenceUnitProperties.JDBC_CONNECTOR and PersistenceUnitProperties.SESSION_EVENT_LISTENER_CLASS properties exactly as mentioned in the following sample code.

```

package com.oracle.mtm.sample;

import com.oracle.microtx.common.MicroTxConfig;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolXADataSource;
import org.eclipse.persistence.config.PersistenceUnitProperties;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import
org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.EclipseLinkJpaVendorAdapter;
import org.eclipse.persistence.jpa.PersistenceProvider;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import javax.sql.XADataSource;

```

```
import java.sql.SQLException;
import java.util.Properties;

@Configuration
@ComponentScan("com.oracle.microtx")
public class XADataSourceConfig {
    @Value("${departmentDataSource.url}")
    private String url;
    @Value("${departmentDataSource.user}")
    private String username;
    @Value("${departmentDataSource.password}")
    private String password;

    @Value("${departmentDataSource.oracleucp.min-pool-size}")
    private String minPoolSize;
    @Value("${departmentDataSource.oracleucp.initial-pool-size:10}")
    private String initialPoolSize;

    @Value("${departmentDataSource.oracleucp.max-pool-size}")
    private String maxPoolSize;

    @Value("${departmentDataSource.oracleucp.data-source-name}")
    private String dataSourceName;

    @Value("${departmentDataSource.oracleucp.connection-pool-name}")
    private String connectionPoolName;

    @Value("${departmentDataSource.oracleucp.connection-factory-class-name:oracle.jdbc.xa.client.OracleXADataSource}")
    private String connectionFactoryClassName;

    @Value("${spring.microtx.xa-resource-manager-id}")
    private String resourceManagerId;

    @Bean(name = "ucpXADataSource")
    @Primary
    public DataSource getDataSource() {
        DataSource pds = null;
        try {
            pds = PoolDataSourceFactory.getPoolXADataSource();

            ((PoolXADataSource)
pds).setConnectionFactoryClassName(connectionFactoryClassName);
            ((PoolXADataSource) pds).setURL(url);
            ((PoolXADataSource) pds).setUser(username);
            ((PoolXADataSource) pds).setPassword(password);
            ((PoolXADataSource)
pds).setMinPoolSize(Integer.valueOf(minPoolSize));
            ((PoolXADataSource)
pds).setInitialPoolSize(Integer.valueOf(initialPoolSize));
            ((PoolXADataSource)
pds).setMaxPoolSize(Integer.valueOf(maxPoolSize));

            ((PoolXADataSource) pds).setDataSourceName(dataSourceName);
            ((PoolXADataSource)
pds).setConnectionPoolName(connectionPoolName);
```

```

        System.out.println("XADataSourceConfig: XADataSource
created");
    } catch (SQLException ex) {
        System.err.println("Error connecting to the database: " +
ex.getMessage());
    }
    return pds;
}

@Bean(name = "entityManagerFactory")
public EntityManagerFactory createEntityManagerFactory() throws
SQLException {
    LocalContainerEntityManagerFactoryBean entityManagerFactoryBean
= new LocalContainerEntityManagerFactoryBean();

    entityManagerFactoryBean.setDataSource(getDataSource());
    entityManagerFactoryBean.setPackagesToScan(new String[]
{ "com.oracle.mtm.sample.entity" });
    entityManagerFactoryBean.setJpaVendorAdapter(new
EclipseLinkJpaVendorAdapter());

entityManagerFactoryBean.setPersistenceProviderClass(PersistenceProvider
.class);
    entityManagerFactoryBean.setPersistenceUnitName("mydeptxads");

    Properties properties = new Properties();
    properties.setProperty( "javax.persistence.transactionType",
"RESOURCE_LOCAL"); // change this to resource_local
    properties.setProperty("javax.persistence.jdbc.driver",
"oracle.jdbc.OracleDriver");
    properties.setProperty("javax.persistence.jdbc.url", url);
    properties.setProperty("javax.persistence.jdbc.user", username);
    properties.setProperty("javax.persistence.jdbc.password",
password);

properties.setProperty(PersistenceUnitProperties.CACHE_SHARED_DEFAULT,
"false");

properties.setProperty(PersistenceUnitProperties.TARGET_DATABASE,
"Oracle");
    properties.setProperty(PersistenceUnitProperties.WEAVING,
"false");

properties.setProperty(PersistenceUnitProperties.JDBC_CONNECTOR,
"com.oracle.microtx.eclipselink.EclipseLinkXADataSourceConnector");

properties.setProperty(PersistenceUnitProperties.SESSION_EVENT_LISTENER_
CLASS,
"com.oracle.microtx.eclipselink.EclipseLinkXASessionEventAdaptor");

    entityManagerFactoryBean.setJpaProperties(properties);
    entityManagerFactoryBean.afterPropertiesSet();
    EntityManagerFactory emf = (EntityManagerFactory)

```

```

entityManagerFactoryBean.getObject();
    System.out.println("entityManagerFactory = " + emf);
    MicroTxConfig.initEntityManagerFactory(emf,
resourceManagerId); // Initialize TMM Library
    return emf;
}
}

```

5. To initialize the Entity Manager Factory object, pass the required parameters to `MicroTxConfig.initEntityManagerFactory()` based on whether your application connects to a single resource manager or multiple resource managers.
 - When your application connects to a single resource manager, create an entity manager factory object and then pass it to the MicroTx Distributed Transactions library. In the following sample code, *emf* is the name of the entity manager factory object.

```
MicroTxConfig.initEntityManagerFactory(emf);
```

- When your application connects with multiple resource managers, you must pass the following parameters while calling `MicroTxConfig.initEntityManagerFactory()`.

```
MicroTxConfig.initEntityManagerFactory(emf, departmentDataSource,
ORCL1-8976-9776-9873);
```

Where,

- *emf* is the entity manager factory object that you have created, and then you pass it to the MicroTx Distributed Transactions library.
 - *departmentDataSource* is the name of the data source that you have created in the above sample code before calling `MicroTxConfig.initEntityManagerFactory()`.
 - *ORCL1-8976-9776-9873* is the resource manager ID (RMID).
6. Insert the following lines in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions client library. The following code in the participant application injects the connection object that is created by the MicroTx Distributed Transactions client library.
 - If you use a single resource manager with a single application, inject an `EntityManager` object as shown in the following code sample, where *microTxEntityManager* is the name of the entity manager factory object.

```

@Autowired
@Qualifier("microTxEntityManager")
@Lazy
private EntityManager entityManager;

```

- When you use multiple resource managers with your application, inject an `EntityManager` object for every resource manager as shown in the following code sample.

```

//Initialize the application context
private ApplicationContext applicationContext;
//Entity Manager object for XA-compliant resource manager 1
private EntityManager entityManager;
//Entity Manager object for XA-compliant resource manager 2

```

```

private EntityManager creditEntityManager;
@Autowired
public AccountService(ApplicationContext applicationContext) {
    this.applicationContext = applicationContext;
    try {
        entityManager = (EntityManager)
applicationContext.getBean("microTxEntityManager",
"departmentDataSource");
        creditEntityManager = (EntityManager)
applicationContext.getBean("microTxEntityManager", "creditDataSource");
    } catch (ClassCastException ex) {
        LOG.info(ex.getMessage());
    }
}

```

Where, *microTxEntityManager* is the entity manager factory object and *departmentDataSource* and *creditDataSource* are data source objects that you have created in the previous step. The earlier code sample provides details about *departmentDataSource*. Provide information for other resource managers, such as *creditDataSource* in a similar way.

Repeat this step for every resource manager.

7. In your application code, inject the entity manager object that you have passed to the MicroTx Distributed Transactions library. Use the entity manager object in your application code based on your business logic, and then use this object to connect to the database.
8. Save the changes.

If there are multiple transaction participant services, then complete these steps for all the participant services.

Source code of a sample Spring REST transaction participant application which uses the MicroTx Distributed Transactions library and Hibernate as the JPA provider is available in the [department-spring-jpa](#) folder. Source code of a sample Spring REST transaction participant application which uses the MicroTx Distributed Transactions library and EclipseLink as the JPA provider is available in the [department-spring-jpa-eclipselink](#) folder. Use the source code, which is available in the [microtx-samples](#) GitHub repository, as a reference while integrating the MicroTx libraries with your application.

3.11 Develop JAX-RS Apps with XA

This section provides information about using the MicroTx Distributed Transactions library with Helidon applications that use the JAX-RS implementation.

For information about using the MicroTx Distributed Transactions library with Spring Boot applications that use the JAX-RS implementation, see [Develop Spring Boot Apps with XA](#).

Topics

- [About MicroTx library for JAX-RS Apps](#)
Integrate the MicroTx Distributed Transactions library with your JAX-RS applications.
- [Configure JAX-RS App as Transaction Initiator](#)
A transaction initiator service initiates or starts a transaction. Based on your application's business logic, a transaction initiator service may only start the transaction or start the transaction and participate in the transaction as well.

- [Configure JAX-RS App as Transaction Participant](#)
Based on whether the resource manager is compliant with XA or not, set environment variables and implement different classes from the MicroTx Distributed Transactions library to configure your participant application.
- [Configure JPA-Based Java App as Transaction Participant](#)
Use the information provided in this section to configure Hibernate or EclipseLink as the JPA provider for your Helidon or Spring Boot applications.

3.11.1 About MicroTx library for JAX-RS Apps

Integrate the MicroTx Distributed Transactions library with your JAX-RS applications.

The MicroTx Distributed Transactions library for JAX-RS performs the following functions:

- Enlists the participant service with the Transaction Coordinator in the transaction.
- Injects an `XDataSource` object for the participant application code to use through dependency injection, and then calls `start()` on the associated `XAResource`. Participant microservices, those microservices called in the context of an XA transaction, must use an XA-compliant data source. In Java this means using an `XDataSource` object. The MicroTx Distributed Transactions libraries automatically inject the configured data source into the participant services, so the application developer must add the `@Inject` or `@Context` annotation to the application code. The application code runs the DML using this connection.
- Calls the resource managers to perform operations.

3.11.2 Configure JAX-RS App as Transaction Initiator

A transaction initiator service initiates or starts a transaction. Based on your application's business logic, a transaction initiator service may only start the transaction or start the transaction and participate in the transaction as well.

Before you begin, identify if your application only initiates the transaction or initiates and participates in the transaction. Configure your application accordingly as the requirements vary slightly for the two scenarios.

Let us consider two scenarios to understand if your application only initiates the transaction or participates in the transaction as well.

- Scenario 1: A banking teller application transfers an amount from one department to another. Here, the teller application only initiates the transaction and does not participate in it. Based on the business logic, the teller application calls different services to complete the transaction. A database instance may or may not be attached to the teller application.
- Scenario 2: A banking teller application transfers an amount from one department to another. For every transaction, the teller application charges 1% as commission. Here, the teller application initiates the transaction and participates in it. A database instance must be attached to the teller application to save the transaction information.

To configure your Java application as a transaction initiator:

1. Specify property values for the MicroTx Distributed Transactions library. See [Configure Library Properties for JAX-RS and Node.js Apps](#).
2. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.

- In Jakarta EE8 environments, such as Helidon 2.x, use the TmmLib file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib</artifactId>
  <version>26.1</version>
</dependency>
```

- In Jakarta EE9 environments, such as Helidon 3.x applications, use the TmmLib-jakarta file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib-jakarta</artifactId>
  <version>26.1</version>
</dependency>
```

3. Import the `oracle.tmm.jta.TrmUserTransaction` package.

```
import oracle.tmm.jta.TrmUserTransaction;
```

4. Optional. Specify the amount of time, in milliseconds, for which the transaction remains active. If a transaction is not committed or rolled back within the specified time period, the transaction is rolled back. This overrides the `oracle.tmm.TransactionTimeout` property value in the application's `tmm.properties` file. See [Set Transaction Timeout](#).

```
transaction.setTransactionTimeout(60000);
```

5. Initialize an object of the `TrmUserTransaction` class for all new transactions to demarcate transaction boundaries in the application code, such as to begin, commit, or roll back transactions. Initialize the object before your application logic initiates or begins a transaction.

The following code sample demonstrates how to create an instance of the `TrmUserTransaction` class called `ut`.

```
UserTransaction ut = new oracle.tmm.jta.TrmUserTransaction();
```

6. The following code samples demonstrate how to begin a new XA transaction called.
 - If your application only initiates the transaction and does not participate in the transaction, add the following line to your application code to begin the XA transaction.

```
ut.begin();
... // Implement the business logic to begin a transaction.
```

- If your application initiates the transaction and participates in it, add the following line to your application code to begin the XA transaction.

```
ut.begin(true);
... // Implement the business logic to begin a transaction.
```

7. Create a REST client.

The following command creates a new client called *svcClient*.

```
Client svcClient = ClientBuilder.newClient();
```

Use this REST client to call the endpoints of the transaction participant services to perform the transaction. The transaction initiator service begins the transaction. To complete the transaction, the initiator service may have to make calls to one or more participant services. While calling the participant services, use the REST client that you have created.

8. Based on your business logic, commit or rollback the transaction.

- To commit a transaction:

```
ut.commit();
```

- To rollback a transaction:

```
ut.rollback();
```

Source code of a sample JAX-RS transaction initiator application which uses the MicroTx Distributed Transactions library and XA is available in the [teller](#) folder which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application. This provides an example of how you can use the MicroTx Distributed Transactions libraries with the business logic of your initiator application. This sample application is called Teller. It initiates a transaction between two departments. It calls Dept A to withdraw an amount and it calls Dept B to deposit the amount. If the initiator service also participates in the transaction in addition to initiating the transaction, you must make additional configurations for the application to participate in the transaction and communicate with the resource manager. See [Configure JAX-RS App as Transaction Participant](#). The [teller-as-participant](#) folder which is located in the `microtx-samples` GitHub repository provides the source code of a JAX-RS transaction initiator application which also participates in the transaction.

3.11.3 Configure JAX-RS App as Transaction Participant

Based on whether the resource manager is compliant with XA or not, set environment variables and implement different classes from the MicroTx Distributed Transactions library to configure your participant application.

- [Configure JAX-RS App with an XA-Compliant Resource Manager](#)
Use the information provided in this section to configure your JAX-RS participant applications when you use an XA-compliant resource manager.
- [Configure JAX-RS App with Multiple XA-Compliant Resource Managers](#)
Use the information provided in this section to configure your JAX-RS participant applications when you use multiple XA-compliant resource managers.
- [Configure JAX-RS App with a Non-XA JDBC Resource](#)
Use the information provided in this section to configure your JAX-RS participant applications when you use a JDBC resource that does not support XA.
- [Configure JAX-RS App with a Non-XA and Non-JDBC Resource](#)
Use the information provided in this section to configure your JAX-RS participant applications when you use a resource that does not support XA and JDBC.

3.11.3.1 Configure JAX-RS App with an XA-Compliant Resource Manager

Use the information provided in this section to configure your JAX-RS participant applications when you use an XA-compliant resource manager.

1. Configure property values for the MicroTx Distributed Transactions client library.

The following example provides sample values for the properties. Provide the values based on your environment.

```
oracle.tmm.TcsConnPoolSize = 15
oracle.tmm.CallbackUrl = https://bookTicket-app:8081
oracle.tmm.PropagateTraceHeaders = true
oracle.tmm.TransactionTimeout = 60000
oracle.tmm.xa.XaSupport = true
```

Ensure that `oracle.tmm.xa.XaSupport` is set to `true`.

For details about each property and other optional properties, see [Configure Library Properties for JAX-RS and Node.js Apps](#).

2. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.

- In Jakarta EE8 environments, such as Helidon 2.x, use the `TmmLib` file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib</artifactId>
  <version>26.1</version>
</dependency>
```

- In Jakarta EE9 environments, such as Helidon 3.x applications, use the `TmmLib-jakarta` file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib-jakarta</artifactId>
  <version>26.1</version>
</dependency>
```

3. Initialize an `XADataSource` object.

The MicroTx Distributed Transactions client library needs to access an `XADataSource` object. It uses this object to create `XAConnection` and `XAResource` objects to connect with a resource manager or database server. The following code describes how you can define the `XADataSource` object at the beginning of the application code when you create the connection object.

```
class oracle.tmm.jta.TrmConfig
static void initXADataSource(XADataSource xaDs)
```

For more information about `XADataSource`, see <https://docs.oracle.com/javase/8/docs/api/javax/sql/XADataSource.html>.

4. In the transaction participant function or block, specify the XADataSource object which is used by the MicroTx Distributed Transactions client library. Provide the credentials and other details to connect to the resource manager.

```
//Example for a participant using an Oracle Database:
OracleXADataSource dataSource = new
oracle.jdbc.xa.client.OracleXADataSource();
dataSource.setURL(url); //database connection string
dataSource.setUser(user); //username to access database
dataSource.setPassword(password); //password to access database
TrmConfig.initXADataSource((XADataSource)dataSource);
```

It is the responsibility of the application developer to ensure that an XA-compliant JDBC driver and required parameters are set up while allocating XADataSource.

The MicroTx Distributed Transactions client library uses the XADataSource object to create database connections.

5. In the transaction participant function or block, add the following line of code only once after you have initialized the XADataSource object.

```
oracle.tmm.jta.TrmConfig.initXADataSource((XADataSource)xaDs);
```

XADataSource is an interface defined in JTA whose implementation is provided by the JDBC driver.

The MicroTx Distributed Transactions client library uses this object to connect to database to start XA transactions and perform various operations such as prepare, commit, and rollback. The MicroTx Distributed Transactions library also provides a SQL connection object to the application code to execute DML using dependency injection.

6. Insert the following line in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions client library. The following code in the participant application injects the connection object that is created by the MicroTx Distributed Transactions client library.

```
@Inject
@TrmSQLConnection
private Connection connection;
```

7. Insert the following lines in the code of the participant service so that the service uses the injected connection object whenever the participant service performs a DML operation.

```
Statement stmt1 = connection.createStatement();
stmt1.execute(query);
stmt1.close();
```

Where, connection is the name of the Connection object that you have injected in the previous step.

Insert these lines of code for every DML operation that your participant service performs. Create a new statement object, such as stmt1 or stmt2 for every DML operation, but use the same connection object that is created by the MicroTx Distributed Transactions client library.

8. Only for Spring Boot applications that use the JAX-RS API, perform the following tasks after registering the resource endpoint that participates in the XA transaction.

- a. Register the filters and XAResourceCallbacks for prepare, commit, rollback as shown in the following sample code snippet.

```
@Component
public class JerseyConfig extends ResourceConfig
{
    public JerseyConfig()
    {
        // Register the MicroTx Distributed Transactions XA resource
        // callback which
        // coordinates with the transaction coordinator
        register(XAResourceCallbacks.class);
        // Register the filters for the MicroTx Distributed
        // Transactions libraries that
        // intercept the JAX_RS calls and manage the XA transactions
        register(TrmTransactionResponseFilter.class);
        register(TrmTransactionRequestFilter.class);

        // Bind the connection
        ...
    }
}
```

- b. If you are using a single resource manager with your Spring Boot application, bind the TrmXAConnectionFactory object to an XAConnection. Later, you will use the TrmXAConnectionFactory object, a MicroTx Distributed Transactions connection factory object, so that MicroTx Distributed Transactions handles the connection.

```
@Component
public class JerseyConfig extends ResourceConfig
{
    public JerseyConfig()
    {
        // Register the filters as shown in the previous step
        ....

        register(new AbstractBinder() {
            @Override
            protected void configure() {
                //Bind the TrmXAConnectionFactory object to an XAConnection
                object
                bindFactory(TrmXAConnectionFactory.class).to(XAConnection.class);
            }
        });
    }
}
```

9. Save the changes.

If there are multiple JAX-RS transaction participant services complete these steps for all the participant services.

Source code of a sample JAX-RS transaction participant application which uses the MicroTx Distributed Transactions library and XA transaction protocol is available in the [department-helidon](#) folder which is located in the microtx-samples GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.11.3.2 Configure JAX-RS App with Multiple XA-Compliant Resource Managers

Use the information provided in this section to configure your JAX-RS participant applications when you use multiple XA-compliant resource managers.

Your application can connect to multiple XA-compliant resource managers. Additionally, your application can connect to a single non-XA resource.

1. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.

- In Jakarta EE8 environments, such as Helidon 2.x, use the `TmmLib` file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib</artifactId>
  <version>26.1</version>
</dependency>
```

- In Jakarta EE9 environments, such as Helidon 3.x applications, use the `TmmLib-jakarta` file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib-jakarta</artifactId>
  <version>26.1</version>
</dependency>
```

2. Create a `DataSourceInfo` object for each resource manager. Ensure that you provide the data source names and resource manager IDs that you have specified in the your application's YAML file.

Sample Command

```
DataSourceInfo departmentDataSourceInfo = new
DataSourceInfo("ORCL1-8976-9776-9873");
departmentDataSourceInfo.setDataSourceName(departmentDataSource);
```

```
DataSourceInfo creditDataSourceInfo = new
DataSourceInfo("ORCL2-2134-5668-8672");
creditDataSourceInfo.setDataSourceName(creditDataSource);
```

Where,

- `departmentDataSource` and `creditDataSource` are names of the XA data source that you have provided in your application's YAML file.
- `ORCL1-8976-9776-9873` and `ORCL2-2134-5668-8672` are the resource manager IDs that you have provided respectively for `departmentDataSource` and `creditDataSource` in your application's YAML file.

Later, you will use the `@Inject` annotation to ensure that your application uses these data sources.

3. Enter only *one* of the following MicroTx Distributed Transactions client library properties for each `DataSourceInfo` object that you have created. The following example provides

property value for the `creditDataSource` object. Similarly, you can provide property values for other resource managers. If you don't provide any value, by default the resource is considered to be XA-compliant.

- For XA-compliant resources, enter:

```
creditDataSource.setXaSupport();
```

- For Oracle RAC database, enter:

```
creditDataSource.setRAC(true);
```

- For non-XA resources that use LLR optimization, enter:

```
creditDataSource.setLLRSupport();
```

- For non-XA resources that use LRC optimization, enter:

```
creditDataSource.setLRCSupport();
```

4. Initialize an `XADatasource` object. If you are using multiple resource managers with your application, initialize the `XADatasource` object in the following way for every XA-compliant resource manager.

The MicroTx Distributed Transactions client library needs to access an `XADatasource` object. It uses this object to create `XAConnection` and `XAResource` objects to connect with a resource manager or database server. The following code describes how you can define the `XADatasource` object at the beginning of the application code when you create the connection object.

```
class oracle.tmm.jta.TrmConfig
static void initXaDataSource(XADatasource dataSource, DataSourceInfo
creditDataSource)
```

Where, `creditDataSource` is the `DataSourceInfo` object that you have previously created.

Repeat this step for every resource manager.

For more information about `XADatasource`, see <https://docs.oracle.com/javase/8/docs/api/javax/sql/XADatasource.html>.

5. In the transaction participant function or block, specify the `XADatasource` object which is used by the MicroTx Distributed Transactions client library. Provide the credentials and other details to connect to the resource manager.

```
//Example for a participant using an Oracle Database:
OracleXADatasource dataSource = new
oracle.jdbc.xa.client.OracleXADatasource();
dataSource.setURL(url); //database connection string
dataSource.setUser(user); //username to access database
dataSource.setPassword(password); //password to access database
```

It is the responsibility of the application developer to ensure that an XA-compliant JDBC driver and required parameters are set up while allocating `XADatasource`.

The MicroTx Distributed Transactions client library uses the `XADatasource` object to create database connections.

Repeat this step for every resource manager.

6. In the transaction participant function or block, add the following line of code only once after you have initialized the XADatasource object.

```
oracle.tmm.jta.TrmConfig.initXaDataSource(dataSource, creditDataSource)
```

Where, *creditDataSource* is the DataSourceInfo object that you have previously created.

The MicroTx Distributed Transactions client library uses this object to connect to database to start XA transactions and perform various operations such as prepare, commit, and rollback. The MicroTx Distributed Transactions library also provides a SQL connection object to the application code to execute DML using dependency injection.

Repeat this step for every resource manager.

7. Only for Spring Boot applications that use the JAX-RS API, perform the following tasks after registering the resource endpoint that participates in the XA transaction.
 - a. Register the filters and XAResourceCallbacks for prepare, commit, rollback as shown in the following sample code snippet.

```
@Component
public class JerseyConfig extends ResourceConfig
{
    public JerseyConfig()
    {
        // Register the MicroTx Distributed Transactions XA resource
        // callback which
        // coordinates with the transaction coordinator
        register(XAResourceCallbacks.class);
        // Register the filters for the MicroTx Distributed
        // Transactions libraries that
        // intercept the JAX_RS calls and manage the XA transactions
        register(TrmTransactionResponseFilter.class);
        register(TrmTransactionRequestFilter.class);

        // Bind the connection
        ...
    }
}
```

- b. Initialize a Bean for each resource manager. The following sample code snippet describes how you can initialize two Beans, one for each resource manager that the application uses. In the following code sample, departmentDataSource and creditDataSource are names of the XA data source that you have provided in your application's YAML file. Note down the names of the data source as you will use the @Inject annotation to ensure that the participant application uses these connections.

```
@Component
public class JerseyConfig extends ResourceConfig
{
    public JerseyConfig()
    {
        // Register the filters as shown in the previous step
        ....
    }
    @Bean
```

```

        @TrmSQLConnection(name = "departmentDataSource")
        @Lazy
        @RequestScope
        public Connection departmentDSSqlConnectionBean(){
            return new
TrmConnectionFactory().getConnection("departmentDataSource");
        }

        @Bean
        @TrmSQLConnection(name = "creditDataSource")
        @Lazy
        @RequestScope
        public Connection creditDSSqlConnectionBean(){
            return new
TrmConnectionFactory().getConnection("creditDataSource");
        }
    }
}

```

8. Insert the following line in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions client library. The following code in the participant application injects the connection object that is created by the MicroTx Distributed Transactions client library.

If you are using multiple resource managers with your application, inject a connection object in the following way for every XA-compliant resource manager.

```

@Inject
@TrmSQLConnection(name = "creditDataSource")
private Connection creditConnection;

```

Where, *creditDataSource* is the value that you have provided for the *dataSourceName* string in the *DataSourceInfo* class of the *oracle.tmm.jta.common.DataSourceInfo* package.

Repeat this step for every resource manager.

9. Insert the following lines in the code of the participant service so that the service uses the injected connection object whenever the participant service performs a DML operation.

```

Statement stmt1 = creditConnection.createStatement();
stmt1.execute(query);
stmt1.close();

```

Where, *creditConnection* is the name of the *Connection* object that you have injected in the previous step.

Insert these lines of code for every DML operation that your participant service performs. Create a new statement object, such as *stmt1* or *stmt2* for every DML operation, but use the same *creditConnection* object that is created by the MicroTx Distributed Transactions client library.

Repeat this step for every resource manager.

10. Save the changes.

If there are multiple JAX-RS transaction participant services complete these steps for all the participant services.

Source code of a sample JAX-RS transaction participant application which uses the MicroTx Distributed Transactions library and XA transaction protocol is available in the [department-helidon-multiplerm](#) folder which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.11.3.3 Configure JAX-RS App with a Non-XA JDBC Resource

Use the information provided in this section to configure your JAX-RS participant applications when you use a JDBC resource that does not support XA.

Your application can connect to multiple XA-compliant resource managers. Additionally, your application can connect to a single non-XA resource.

1. When you use a single resource manager, provide values for all the MicroTx Distributed Transactions client library properties in a single file, such as `tmm.properties` file. Skip this step if you are using multiple resource managers for your application.

Ensure that `oracle.tmm.xa.XaSupport` is set to `false` and `oracle.tmm.xa.LLRSupport` or `oracle.tmm.xa.LRCSupport` is set to `true`.

- To enable the Logging Last Resource (LLR) optimization, set the following values for the environment variables.

```
oracle.tmm.xa.XaSupport = false
oracle.tmm.xa.LLRSupport = true
oracle.tmm.xa.LRCSupport = false
oracle.tmm.TcsConnPoolSize = 15
oracle.tmm.CallbackUrl = https://bookHotel-app:8081
oracle.tmm.PropagateTraceHeaders = true
oracle.tmm.TransactionTimeout = 60000
```

- To enable the Last Resource Commit (LRC) optimization, set the following values for the environment variables.

```
oracle.tmm.xa.XaSupport = false
oracle.tmm.xa.LLRSupport = false
oracle.tmm.xa.LRCSupport = true
oracle.tmm.TcsConnPoolSize = 15
oracle.tmm.CallbackUrl = https://bookHotel-app:8081
oracle.tmm.PropagateTraceHeaders = true
oracle.tmm.TransactionTimeout = 60000
```

2. If you are using a multiple resource managers with your application, complete the following steps to configure property values for the MicroTx Distributed Transactions client library. Skip this step if you are using a single resource manager.

- a. Create a `DataSourceInfo` object for each resource manager. Ensure that you provide the data source names and resource manager IDs that you have the specified in the your application's YAML file.

Sample Command

```
DataSourceInfo departmentDataSourceInfo = new
DataSourceInfo("ORCL1-8976-9776-9873");
departmentDataSourceInfo.setDataSourceName(departmentDataSource);
```

```
DataSourceInfo creditDataSourceInfo = new
```

```
DataSourceInfo("ORCL2-2134-5668-8672");
creditDataSourceInfo.setDataSourceName(creditDataSource);
```

Where,

- departmentDataSource and creditDataSource are names of the XA data source that you have provided in your application's YAML file.
- ORCL1-8976-9776-9873 and ORCL2-2134-5668-8672 are the resource manager IDs that you have provided respectively for departmentDataSource and creditDataSource in your application's YAML file.

Later, you will use the @Inject annotation to ensure that your application uses these data sources.

- b. Enter only *one* of the following MicroTx Distributed Transactions client library properties for each DataSourceInfo object that you have created. The following example provides property value for the creditDataSource object. Similarly, you can provide property values for other resource managers. If you don't provide any value, by default the resource is considered to be XA-compliant.
 - For XA-compliant resources, enter creditDataSource.setXaSupport();.
 - For non-XA resources that use LLR optimization, enter creditDataSource.setLLRSupport();.
 - For non-XA resources that use LRC optimization, enter creditDataSource.setLRCSupport();.
3. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.
 - In Jakarta EE8 environments, such as Helidon 2.x, use the TmmLib file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib</artifactId>
  <version>26.1</version>
</dependency>
```

- In Jakarta EE9 environments, such as Helidon 3.x applications, use the TmmLib-jakarta file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib-jakarta</artifactId>
  <version>26.1</version>
</dependency>
```

4. Enable session affinity. See [Enable Session Affinity](#).
5. Initialize a Datasource object.

The MicroTx Distributed Transactions library needs to access a data source object. It uses the data source object to create java.sql.Connection objects to connect with a resource manager. The following code describes how you can define a data source object.

You must provide this code at the start of the application, so that the initNonXaDataSource method is called immediately after the server starts and before any other requests are served.

- If you are using a single resource manager with your application, initialize a data source in the following way.

```
class oracle.tmm.jta.TrmConfig
static void initNonXaDataSource(DataSource NonXaDs)
```

- If you are using multiple resource managers with your application, initialize the data source object in the following way for the Non-XA JDBC resource. A participant service can connect to multiple XA-compliant resource managers, but only one non-XA resource is supported in a transaction.

```
class oracle.tmm.jta.TrmConfig
static void initNonXaDataSource(DataSource departmentDataSource,
DataSourceInfo departmentDataSourceInfo)
```

Where, *dataSourceInfo* is the object that you have created in the step 2.

6. In the transaction participant function or block, specify the *DataSource* object which is used by the MicroTx Distributed Transactions library. Provide the credentials and database driver details to connect to the resource manager. The following example shows the details that you must provide when you use MySQL database as an LLR. Similarly, you can provide credentials and database driver information for other databases.

```
//Example for a participant using a MySQL database as resource manager
this.dataSource = PoolDataSourceFactory.getPoolDataSource();
this.dataSource.setURL(url); //Database connection string
this.dataSource.setUser(user); //User name to access the database
this.dataSource.setPassword(password); //Password to access the database
//Database driver information for the MySQL database.
//Provide the JDBC driver information that is specific to your database.
this.dataSource.setConnectionFactoryClassName("com.mysql.cj.jdbc.MysqlDataS
ource");
this.dataSource.setMaxPoolSize(15);
```

It is the application developer's responsibility to ensure that a database-specific JDBC driver and required parameters are set up while allocating *DataSource*.

MicroTx Distributed Transactions library uses the *DataSource* object to create database connections.

7. In the transaction participant function or block, add the following line of code only once after you have initialized the *Datasource* object. The MicroTx Distributed Transactions library uses this object to start a database transaction. The MicroTx Distributed Transactions library also provides a SQL connection object to the application code to execute DML using dependency injection.

```
oracle.tmm.jta.TrmConfig.initNonXaDataSource((DataSource) NonXaDs);
```

Where, *Datasource* is an interface defined in JTA whose implementation is provided by the JDBC driver.

8. Insert the following line in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions library. The following code

in the participant application injects the connection object that is created by the MicroTx Distributed Transactions library.

```
@Inject @TrmNonXASQLExceptionConnection private Connection connection;
```

9. Insert code in the participant service so that the service uses the injected connection object whenever the participant service performs a DML operation. You can create code to use the injected connection object based on your business scenario. Here's an example code snippet.

```
Statement stmt1 = connection.createStatement();
stmt1.execute(query);
stmt1.close();
```

Insert these lines of code for every DML operation that your participant service performs. Create a new statement object, such as stmt1 or stmt2 for every DML operation, but use the same connection object that's created by the MicroTx Distributed Transactions library.

10. Only for Spring Boot applications that use the JAX-RS API, register the XAResource callbacks, such as prepare, commit, rollback, and various filters as shown in the following sample code snippet.

```
@Component
public class JerseyConfig extends ResourceConfig
{
    public JerseyConfig()
    {
        register(XAResourceCallbacks.class);
        register(TrmTransactionResponseFilter.class);
        register(TrmTransactionRequestFilter.class);
        register(new AbstractBinder() {
            @Override
            protected void configure() {

                bindFactory(TrmXAConnectionFactory.class).to(XAConnection.class);
            }
        });
    }
}
```

This is in addition to registering the resource endpoint that participates in the XA transaction.

11. Save the changes.

Source code of a sample JAX-RS transaction participant application which uses the MicroTx Distributed Transactions library and XA transaction protocol is available in the [department-nonxads](#) folder which is located in the microtx-samples GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.11.3.4 Configure JAX-RS App with a Non-XA and Non-JDBC Resource

Use the information provided in this section to configure your JAX-RS participant applications when you use a resource that does not support XA and JDBC.

Your application can connect to multiple XA-compliant resource managers. However, only a single non-XA resource can participate in a transaction.

1. Before you begin, ensure that you have configured the property values for the MicroTx Distributed Transactions library. See [Configure Library Properties for JAX-RS and Node.js Apps](#).

Ensure that `oracle.tmm.xa.XaSupport` is set to `false` and `oracle.tmm.xa.LLRSupport` or `oracle.tmm.xa.LRCSupport` is set to `true`.

- To enable the Logging Last Resource (LLR) optimization, set the following values for the environment variables.

```
oracle.tmm.xa.XaSupport = false
oracle.tmm.xa.LLRSupport = true
oracle.tmm.xa.LRCSupport = false
```

- To enable the Last Resource Commit (LRC) optimization, set the following values for the environment variables.

```
oracle.tmm.xa.XaSupport = false
oracle.tmm.xa.LLRSupport = false
oracle.tmm.xa.LRCSupport = true
```

2. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.

- In Jakarta EE8 environments, such as Helidon 2.x, use the `TmmLib` file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib</artifactId>
  <version>26.1</version>
</dependency>
```

- In Jakarta EE9 environments, such as Helidon 3.x applications, use the `TmmLib-jakarta` file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib-jakarta</artifactId>
  <version>26.1</version>
</dependency>
```

3. Enable session affinity. See [Enable Session Affinity](#).
4. Implement the `NonXAResource` interface.

```
public class MongoDbNonXAResource implements NonXAResource {
    // Provide application-specific code for all the methods in the
```

```
NonXAResource interface.
}
```

For information about the NonXAResource interface, see [Transaction Manager for Microservices Java API Reference](#).

If you have enabled the LRC optimization, you don't have to implement the `recover()` method in the NonXAResource interface as the `commit()` method returns NULL for `commitRecord` in LRC.

5. After implementing the NonXAResource interface, import the MicroTx Distributed Transactions library files, and then produce a non-XA resource. Annotate the non-XA resource that you create with `@NonXa` annotation. The MicroTx Distributed Transactions library consumes the object that you annotate.

The following example shows a sample implementation for a MongoDB resource. Create code for your application based on your business requirements. In this example, the `NonXaResourceFactory` class supplies the NonXAResource. It produces a non-XA resource, and then the MicroTx Distributed Transactions library consumes the non-XA resource.

```
package com.oracle.mtm.sample.nonxa;

import oracle.tmm.jta.nonxa.NonXAResource;
import oracle.tmm.jta.nonxa.NonXa;

import javax.enterprise.inject.Produces;
import javax.inject.Inject;
import javax.ws.rs.ext.Provider;
import java.util.function.Supplier;

@Provider
public class NonXaResourceFactory implements Supplier<NonXAResource> {

    @Inject
    MongoDBNonXAResource nonXAResource;

    @Produces
    @NonXa
    public NonXAResource getNonXAResource() {
        return nonXAResource;
    }

    @Override
    public NonXAResource get() {
        return getNonXAResource();
    }
}
```

6. Save the changes.

Source code of a sample JAX-RS transaction participant application which uses the MicroTx Distributed Transactions library and XA transaction protocol is available in the [department-nonxa](#) folder which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.11.4 Configure JPA-Based Java App as Transaction Participant

Use the information provided in this section to configure Hibernate or EclipseLink as the JPA provider for your Helidon or Spring Boot applications.

Configuring a JPA-Based Java app as a transaction participant is similar to configuring a JDBC-based Java app as a transaction participant.

To configure a JDBC-based Java app as a transaction participant, create a custom data source object and then pass this object to the MicroTx Distributed Transactions library. In your Java application code, inject a `TrmSQLConnection` connection object with details of the custom data source, and then update the application code to use the injected object.

To configure a JPA-based Java app as a transaction participant, create a custom data source object and then pass this object to the MicroTx Distributed Transactions library. In your application code, inject a `TrmEntityManager` object with details of the custom data source, and then update the application code to use the injected object.

- [Configure JPA-Based Java App with an XA-Compliant Resource Manager](#)
Use the information provided in this section to configure Hibernate or EclipseLink as the JPA provider for your Helidon or Spring Boot applications which participates in an XA transaction when you use an XA-compliant resource manager.

3.11.4.1 Configure JPA-Based Java App with an XA-Compliant Resource Manager

Use the information provided in this section to configure Hibernate or EclipseLink as the JPA provider for your Helidon or Spring Boot applications which participates in an XA transaction when you use an XA-compliant resource manager.

Your application can connect to multiple XA-compliant resource managers. If you are using multiple XA-compliant resource managers for your application, complete the following steps for each resource manager.

1. Configure property values for the MicroTx Distributed Transactions client library properties.

The following example provides sample values for the properties. Provide the values based on your environment.

```
oracle.tmm.TcsConnPoolSize = 15
oracle.tmm.CallbackUrl = https://bookTicket-app:8081
oracle.tmm.PropagateTraceHeaders = true
oracle.tmm.TransactionTimeout = 60000
oracle.tmm.xa.XaSupport = true
```

Ensure that `oracle.tmm.xa.XaSupport` is set to `true`.

For details about each property and other optional properties, see [Configure Library Properties for JAX-RS and Node.js Apps](#).

2. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.
 - In Jakarta EE8 environments, such as Helidon 2.x, use the `TmmLib` file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib</artifactId>
```

```
<version>26.1</version>
</dependency>
```

- In Jakarta EE9 environments, such as Helidon 3.x applications, use the TmmLib-jakarta file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib-jakarta</artifactId>
  <version>26.1</version>
</dependency>
```

3. Perform this task only for Spring Boot applications that use the JAX-RS API. Create a .java file in the folder that contains your application code to initialize an XADataSourceConfig object. The XADataSourceConfig class contains methods to create custom data source and entity manager factory objects.

The following example code shows how you can initialize the library in within the XADataSourceConfig class, create a custom data source named departmentDataSource, and create an entity manager factory object named emf. You can create a similar code for your application.

The custom data source object contains details to connect with the resource manager. It is the responsibility of the application developer to ensure that an XA-compliant JDBC driver and required parameters are set up while creating a custom data source object.

```
package com.oracle.tmm.sample;

import oracle.tmm.common.TrmConfig;
import
oracle.tmm.jta.jpa.hibernate.HibernateXADataSourceConnectionProvider;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolXADataSource;
import org.hibernate.jpa.HibernatePersistenceProvider;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import
org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.sql.SQLException;
import java.util.Properties;

@Configuration
@EnableTransactionManagement
public class XADataSourceConfig {
    @Value("${spring.xads.datasource.url}")
    private String url;
    @Value("${spring.xads.datasource.username}")
    private String username;
    @Value("${spring.xads.datasource.password}")
    private String password;
```

```

@Value("${spring.xads.datasource.oracleucp.min-pool-size}")
private String minPoolSize;
@Value("${spring.xads.datasource.oracleucp.initial-pool-size:10}")
private String initialPoolSize;

@Value("${spring.xads.datasource.oracleucp.max-pool-size}")
private String maxPoolSize;

@Value("${spring.xads.datasource.oracleucp.data-source-name}")
private String dataSourceName;

@Value("${spring.xads.datasource.oracleucp.connection-pool-name}")
private String connectionPoolName;

@Value("${spring.xads.datasource.oracleucp.connection-factory-class-
name:oracle.jdbc.xa.client.OracleXADataSource}")
private String connectionFactoryClassName;

//Create a custom data source object. Provide credentials and other
details to connect to the resource manager.
@Bean(name = "departmentDataSource")
@Primary
public DataSource getDataSource() {
    DataSource pds = null;
    try {
        pds = PoolDataSourceFactory.getPoolXADataSource();

        ((PoolXADataSource)
pds).setConnectionFactoryClassName(connectionFactoryClassName);
        ((PoolXADataSource) pds).setURL(url);
        ((PoolXADataSource) pds).setUser(username);
        ((PoolXADataSource) pds).setPassword(password);
        ((PoolXADataSource)
pds).setMinPoolSize(Integer.valueOf(minPoolSize));
        ((PoolXADataSource) pds).setInitialPoolSize(10);
        ((PoolXADataSource)
pds).setMaxPoolSize(Integer.valueOf(maxPoolSize));

        ((PoolXADataSource) pds).setDataSourceName(dataSourceName);
        ((PoolXADataSource)
pds).setConnectionPoolName(connectionPoolName);

        System.out.println("XADataSourceConfig: XADataSource created");
    } catch (SQLException ex) {
        System.err.println("Error connecting to the database: " +
ex.getMessage());
    }
    return pds;
}

// Create an entity manager factory object
@Bean(name = "entityManagerFactory")
public EntityManagerFactory createEntityManagerFactory() throws
SQLException {
    LocalContainerEntityManagerFactoryBean entityManagerFactoryBean =
new LocalContainerEntityManagerFactoryBean();

```

```

        entityManagerFactoryBean.setDataSource(getDataSource());
        entityManagerFactoryBean.setPackagesToScan(new String[]
{ "com.oracle.mtm.sample.entity" });
        entityManagerFactoryBean.setJpaVendorAdapter(new
HibernateJpaVendorAdapter());

entityManagerFactoryBean.setPersistenceProviderClass(HibernatePersistencePr
ovider.class);
        entityManagerFactoryBean.setPersistenceUnitName("mydeptxads");
        Properties properties = new Properties();
        properties.setProperty( "javax.persistence.transactionType",
"RESOURCE_LOCAL"); // change this to resource_local
        properties.put("hibernate.show_sql", "true");
        properties.put("hibernate.dialect",
"org.hibernate.dialect.Oracle12cDialect");
        properties.put("hibernate.format_sql", "true");
        properties.put("hbm2ddl.auto", "validate");
        properties.put("hibernate.connection.provider_class",
"oracle.tmm.jta.jpa.hibernate.HibernateXADataSourceConnectionProvider");
        entityManagerFactoryBean.setJpaProperties(properties);
        entityManagerFactoryBean.afterPropertiesSet();
        EntityManagerFactory emf = (EntityManagerFactory)
entityManagerFactoryBean.getObject();
        System.out.println("entityManagerFactory = " + emf);
        // Pass the entity manager factory object to the MicroTx Library

        // If you are using a single resource manager with your
application,
        //pass the entity manager factory object to the MicroTx
Distributed Transactions library in the following way.
        TrmConfig.initEntityManagerFactory(emf);
        // If you are using multiple resource managers with your
application,
        // pass the entity manager factory object to the MicroTx
Distributed Transactions library in the following way.
        TrmConfig.initEntityManagerFactory(emf, departmentDataSource,
ORCL1-8976-9776-9873);

        return emf;
    }
}

```

To initialize the Entity Manager Factory object, pass the required parameters to `TrmConfig.initEntityManagerFactory()` based on whether your application connects to a single resource manager or multiple resource managers.

- When your application connects to a single resource manager, create an entity manager factory object and then pass it to the MicroTx Distributed Transactions library. In the following sample code, *emf* is the name of the entity manager factory object.

```
TrmConfig.initEntityManagerFactory(emf);
```

- When your application connects with multiple resource managers, you must pass the following parameters while calling `TrmConfig.initEntityManagerFactory()`.

```
TrmConfig.initEntityManagerFactory(emf, departmentDataSource,
ORCL1-8976-9776-9873);
```

Where,

- `emf` is the entity manager factory object that you have created, and then you pass it to the MicroTx Distributed Transactions library.
 - `departmentDataSource` is the name of the data source that you have created in the above sample code before calling `TrmConfig.initEntityManagerFactory()`.
 - `ORCL1-8976-9776-9873` is the resource manager ID (RMID).
4. Only for Spring Boot applications that use the JAX-RS API, perform the following tasks after registering the resource endpoint that participates in the XA transaction.
 - a. Register the filters and `XAResourceCallbacks` for prepare, commit, rollback as shown in the following sample code snippet.

```
@Component
public class Configuration extends ResourceConfig
{
    public Configuration()
    {
        // Register the MicroTx Distributed Transactions XA resource
        // callback which
        // coordinates with the transaction coordinator
        register(XAResourceCallbacks.class);
        // Register the filters for the MicroTx Distributed
        // Transactions libraries that
        // intercept the JAX_RS calls and manage the XA transactions
        register(TrmTransactionResponseFilter.class);
        register(TrmTransactionRequestFilter.class);

        // Bind the connection
        ...
    }
}
```

- b. Skip this step if you are using multiple resource managers. If you are using a single resource manager with your Spring Boot application, bind the `TrmEntityManager` object to an `EntityManager`. Later, you will use the `TrmEntityManager` object in your application code so that MicroTx Distributed Transactions handles the connection.

```
@Component
public class Configuration extends ResourceConfig
{
    public Configuration()
    {
        // Register the filters as shown in the previous step
        ....
        // Bind the connection
        register(new AbstractBinder() {
            @Override
```

```

        protected void configure() {
            //Bind the TrmEntityManager object to an EntityManager
            object

            bindFactory(TrmEntityManagerFactory.class).to(EntityManager.class);
        }
    });
}
}

```

- c. Skip this step if you are using single resource manager. If you are using multiple resource managers, initialize a Bean for each resource manager. The following sample code snippet describes how you can initialize two Beans, one for each resource manager that the application uses. In the following code sample, `departmentDataSource` and `creditDataSource` are names of the XA data source that you have provided in your application's YAML file. Note down the names of the data source as you will use the `@Inject` annotation to ensure that the participant application uses these connections.

```

@Component
public class Configuration extends ResourceConfig
{
    public Configuration()
    {
        // Register the filters as shown in the previous step
        ....
        // Initialize a bean for every resource manager that you want to
        use with your app
        @Bean
        @TrmEntityManager(name = "departmentDataSource")
        @Lazy
        @RequestScope
        public EntityManager departmentDSSqlConnectionBean() throws
        SQLException {
            return new
            TrmEntityManagerFactory().getEntityManagerByName("departmentDataSource")
            ;
        }

        @Bean
        @TrmEntityManager(name = "creditDataSource")
        @Lazy
        @RequestScope
        public EntityManager creditDSSqlConnectionBean() throws
        SQLException {
            return new
            TrmConnectionFactory().getConnection("creditDataSource");
        }
    }
}

```

5. Create a `.java` file in the folder that contains your application code to initialize an `PoolXADataSource` object. The `PoolXADataSource` class contains methods to create custom data source and entity manager factory objects.

The following example code shows how you can initialize the library in within the PoolXADatasource class, create a custom data source named departmentDataSource, and create an entity manager factory object. You can create similar code for your application.

The custom data source object contains details to connect with the resource manager. It is the responsibility of the application developer to ensure that an XA-compliant JDBC driver and required parameters are set up while creating a custom data source object.

```
package com.oracle.mtm.sample;

import oracle.tmm.common.TrmConfig;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolXADatasource;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.context.Initialized;
import javax.enterprise.event.Observes;
import javax.enterprise.inject.Produces;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import java.lang.invoke.MethodHandles;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.Map;

@ApplicationScoped
public class Configuration {

    private PoolXADatasource dataSource;
    final static Logger logger =
        LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());

    private EntityManagerFactory entityManagerFactory;

    @Inject
    @ConfigProperty(name = "departmentDataSource.url")
    String url;
    @Inject
    @ConfigProperty(name = "departmentDataSource.user")
    String user;
    @Inject
    @ConfigProperty(name = "departmentDataSource.password")
    String password;

    private void init(@Observes @Initialized(ApplicationScoped.class)
        Object event) {
        initialiseDataSource();
        createEntityManagerFactory();
    }
}
```

```

/**
 * Initializes the datasource into the MicroTx library that manages
the lifecycle of the XA transaction
 *
 */
private void initialiseDataSource() {
    try {
        this.dataSource = PoolDataSourceFactory.getPoolXADataSource();
        this.dataSource.setURL(url);
        this.dataSource.setUser(user);
        this.dataSource.setPassword(password);

        this.dataSource.setConnectionFactoryClassName("oracle.jdbc.xa.client.Oracle
XADataSource");
        this.dataSource.setMaxPoolSize(15);
    } catch (SQLException e) {
        logger.error("Failed to initialise database");
    }
}

public PoolXADataSource getDatasource() {
    return dataSource;
}

public void createEntityManagerFactory(){
    Map<String, Object> props = new HashMap<String, Object>();

    props.put("hibernate.connection.datasource", getDatasource());
    props.put("hibernate.show_sql", "true");
    props.put("hibernate.dialect",
"org.hibernate.dialect.Oracle12cDialect");
    props.put("hibernate.hbm2ddl.auto", "none");
    props.put("hibernate.format_sql", "true");
    props.put("hibernate.connection.provider_class",
"oracle.tmm.jta.jpa.hibernate.HibernateXADataSourceConnectionProvider");
    props.put("javax.persistence.transactionType", "RESOURCE_LOCAL");
    props.put("javax.persistence.jdbc.driver",
"oracle.jdbc.OracleDriver");
    props.put("javax.persistence.jdbc.url", url);
    props.put("javax.persistence.jdbc.user", user);
    props.put("javax.persistence.jdbc.password", password);

    this.entityManagerFactory =
Persistence.createEntityManagerFactory("mydeptxads", props);

    //Initialize the MicroTx Library

    // If you are using a single resource manager with your
application,
    //pass the entity manager factory object to the MicroTx library in
the following way.
    TrmConfig.initEntityManagerFactory(this.entityManagerFactory);

    //If you are using multiple resource managers with your
application,
    //set the setRAC property if you are using an Oracle RAC Database

```

```

        departmentDataSource.setRAC(true);

        //If you are using multiple resource managers with your
        application,
        //pass the entity manager factory object to the MicroTx library in
        the following way.
        TrmConfig.initEntityManagerFactory(this.entityManagerFactory,
        departmentDataSource, ORCL1-8976-9776-9873);
    }

    public EntityManagerFactory getEntityManagerFactory() {
        return this.entityManagerFactory;
    }

    public Logger getLogger() {
        return logger;
    }

    /**
     * EntityManager bean for non-distributed database operations.
     */
    @Produces
    public EntityManager getEntityManager(){
        EntityManager entityManager = null;
        try {
            entityManager =
this.getEntityManagerFactory().createEntityManager();
        } catch (RuntimeException e){
            e.printStackTrace();
            logger.error("Entity manager bean for local transactions
creation failed!");
        }
        return entityManager;
    }
}

```

To initialize the Entity Manager Factory object, pass the required parameters to `TrmConfig.initEntityManagerFactory()` based on whether your application connects to a single resource manager or multiple resource managers.

- When your application connects to a single resource manager, create an entity manager factory object and then pass it to the MicroTx Distributed Transactions library.

```
TrmConfig.initEntityManagerFactory(this.entityManagerFactory);
```

- When your application connects with multiple resource managers, you must pass the following parameters while calling `TrmConfig.initEntityManagerFactory()`.

```
TrmConfig.initEntityManagerFactory(this.entityManagerFactory,
departmentDataSource, ORCL1-8976-9776-9873);
```

Where,

- *departmentDataSource* is the name of the data source that you have created in the above sample code before calling `TrmConfig.initEntityManagerFactory()`.

- `ORCL1-8976-9776-9873` is the resource manager ID (RMID).

If you are using multiple resource managers, you must set `departmentDataSource.setRAC(true)` for the data source that uses an Oracle RAC Database.

6. Insert the following line in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions client library. The following code in the participant application injects the connection object that is created by the MicroTx Distributed Transactions client library.
 - If you use a single resource manager with a single application, inject an `EntityManager` object as shown in the following code sample.

```
@Inject
@TrmEntityManager
private EntityManager emf;
```

- When you use multiple resource managers with your application, inject an `EntityManager` object for every resource manager as shown in the following code sample.

```
@Inject
@TrmEntityManager(name = "departmentDataSource")
private EntityManager emf;

@Inject
@TrmEntityManager(name = "creditDataSource")
private EntityManager emf;
```

Where, `emf` is the entity manager factory object and `departmentDataSource` and `creditDataSource` are data source objects that you have created in the previous step. The earlier code sample provides details about `departmentDataSource`. Provide information for other resource managers, such as `creditDataSource` in a similar way.

7. In your application code, inject the entity manager object that you have passed to the MicroTx Distributed Transactions library. Use the entity manager object in your application code based on your business logic, and then use this object to connect to the database.

The following example code shows how the entity manager object is injected and used.

```
@POST
@Path("/{accountId}/withdraw")
public Response withdraw(@PathParam("accountId") String accountId,
    @QueryParam("amount") double amount, @Context EntityManager entityManager)
{
    // Application code or business logic
    if(amount == 0){
        return Response.status(422,"Amount must be greater than
zero").build();
    }
    try {
        if (this.accountService.getBalance(accountId, entityManager) <
amount) {
            return Response.status(422, "Insufficient balance in the
account").build();
        }
    }
}
```

```

        if(this.accountService.withdraw(accountId, amount,
entityManager)) {
            config.getLogger().log(Level.INFO, amount + " withdrawn
from account: " + accountId);
            return Response.ok("Amount withdrawn from the
account").build();
        }
    } catch (SQLException | IllegalArgumentException e) {
        config.getLogger().log(Level.SEVERE, e.getLocalizedMessage());
        return
Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
    }
    return Response.serverError().entity("Withdraw failed").build();
}

```

8. Save the changes.

If there are multiple transaction participant services, then complete these steps for all the participant services.

Source code of a sample JAX-RS transaction participant application which uses the MicroTx Distributed Transactions library and Hibernate as the JPA provider is available in the [department-helidon-jpa](#) folder. Source code of a sample JAX-RS transaction participant application which uses the MicroTx Distributed Transactions library and EclipseLink as the JPA provider is available in the [department-helidon-jpa-eclipselink](#) folder. Use the source code, which is available in the [microtx-samples](#) GitHub repository, as a reference while integrating the MicroTx libraries with your application.

3.12 Configure Java Apps to Leverage Transactional Event Queues

Integrate Oracle MicroTx Distributed Transactions library with your application to leverage the Oracle Transactional Event Queues (TEQ) feature.

The Oracle Transactional Event Queues (TEQ) feature is available with Oracle Database. It provides database-integrated message queuing functionality. See Transactional Event Queues Leverage Oracle Database in *Transactional Event Queues and Advanced Queuing User's Guide*.

TEQ has a publisher and a subscriber. Integrate the MicroTx Distributed Transactions library files with the publisher.

Application developers use a custom annotation, `MicroTxXATopicSession`, provided by the MicroTx Distributed Transactions library to inject a TEQ session and use it in their application code. The MicroTx Distributed Transactions maintains the session lifecycle, which includes the initializing the session, attaching the session to an ongoing XA transaction to perform XA operations, and terminating the session.

- [Required Privileges](#)
Grant the required privileges to the database user to leverage the Oracle Transactional Event Queues (TEQ) feature.
- [Configure JAX-RS Participant App to Leverage Transactional Event Queues](#)
Use the information provided in this section to integrate Oracle MicroTx Distributed Transactions library with your JAX-RS participant application to leverage the Oracle Transactional Event Queues (TEQ) feature.

- [Configure Spring REST Participant App to Leverage Transactional Event Queues](#)
Use the information provided in this section to integrate Oracle MicroTx Distributed Transactions library with your Spring REST participant application to leverage the Oracle Transactional Event Queues (TEQ) feature.

3.12.1 Required Privileges

Grant the required privileges to the database user to leverage the Oracle Transactional Event Queues (TEQ) feature.

- Database user must have the ENQUEUE_ANY and DEQUEUE_ANY privileges to enqueue or dequeue messages through the existing database connection.
- Database user must have EXECUTE privilege on the DBMS_AQIN and DBMS_AQ packages to use the Oracle JMS interfaces.

Sample Command

The following sample command demonstrates how the required privileges are granted to the Oracle Database user, `department_teq`.

```
-- Oracle transaction event Queue(TEQ) related grants
GRANT EXECUTE ON DBMS_AQ TO department_teq;
GRANT EXECUTE ON DBMS_AQIN TO department_teq;

EXECUTE DBMS_AQADM.CREATE_SHARDED_QUEUE (queue_name =>
'department_teq.<queue_name>', multiple_consumers => true);
EXECUTE dbms_aqadm.grant_queue_privilege('ALL',
'department_teq.<queue_name>', 'department_teq');
EXECUTE dbms_aqadm.start_queue(queue_name => 'department_teq.<queue_name>');
```

3.12.2 Configure JAX-RS Participant App to Leverage Transactional Event Queues

Use the information provided in this section to integrate Oracle MicroTx Distributed Transactions library with your JAX-RS participant application to leverage the Oracle Transactional Event Queues (TEQ) feature.

You can also perform the following steps for a JAX-RS application that initiates and participates in an XA transaction.

1. Open the `application.yaml` file in any text editor and provide property values for the MicroTx Distributed Transactions client library and the database details.
 - In addition to the existing data source configuration details, enter a unique TEQ topic name for the session.

```
jms:
  topicName: "MicroTxXATopicSession"
```

Remember this name as you will use it later to inject a `TopicSession` object. If you are using multiple resource managers, enter these details in the `application.yaml` file for each resource manager.

- Provide details to connect to the Oracle Database where TEQ is available. The following example provides sample database URL, password, user name, and RMID

values for the departmentDataSource database. For description of the properties, see Data Store Properties.

```
departmentDataSource:
  url: "jdbc:oracle:thin:@tcps://<host>:<port>/<service_name>?"
  wallet_location=Database_Wallet"
  user: "dbuser"
  password: "xxxxxx"
  rmid: "174A5FF2-D8B1-47B0-AF09-DA5AFECA2F61"
```

Only if you use multiple resource managers, specify a unique resource manager ID (RMID) for each resource manager. Also specify one of these RMIDs in the MicroTx library properties. If you use a single resource manager, you don't need to mention the RMID in the application.yaml file.

- The following example provides sample values for the MicroTx Distributed Transactions client library properties. Provide the values based on your environment.

```
oracle.tmm.TcsConnPoolSize = 15
oracle.tmm.CallbackUrl = https://bookTicket-app:8081
oracle.tmm.PropagateTraceHeaders = true
oracle.tmm.TransactionTimeout = 60000
oracle.tmm.xa.XaSupport = true
```

Ensure that oracle.tmm.xa.XaSupport is set to true.

For details about each property and other optional properties, see [Configure Library Properties for JAX-RS and Node.js Apps](#).

2. Import the oracle.tmm.jta.common.MicroTxXATopicSession package.

```
import oracle.tmm.jta.common.MicroTxXATopicSession;
```

3. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.

```
<dependency>
  <groupId>jakarta.jms</groupId>
  <artifactId>jakarta.jms-api</artifactId>
  <version>3.1.0</version>
</dependency>

<dependency>
  <groupId>com.oracle.database.messaging</groupId>
  <artifactId>aqapi-jakarta</artifactId>
  <version>26.1</version>
</dependency>
```

4. Initialize an XADatasource object.

The MicroTx Distributed Transactions client library needs to access an XADatasource object. It uses this object to create XAConnection and XAResource objects to connect with a resource manager or database server. The following code describes how you can define

the XADataSource object at the beginning of the application code when you create the connection object.

```
class oracle.tmm.jta.TrmConfig
static void initXADataSource(XADataSource xaDs)
```

For more information about XADataSource, see <https://docs.oracle.com/javase/8/docs/api/javax/sql/XADataSource.html>.

5. In the transaction participant function or block, specify the XADataSource object which is used by the MicroTx Distributed Transactions client library. Provide the credentials and other details to connect to the resource manager.

```
//Example for a participant using an Oracle Database:
OracleXADataSource dataSource = new
oracle.jdbc.xa.client.OracleXADataSource();
dataSource.setURL(url); //database connection string
dataSource.setUser(user); //username to access database
dataSource.setPassword(password); //password to access database
TrmConfig.initXADataSource((XADataSource)dataSource);
```

It is the responsibility of the application developer to ensure that an XA-compliant JDBC driver and required parameters are set up while allocating XADataSource.

The MicroTx Distributed Transactions client library uses the XADataSource object to create database connections.

6. In the transaction participant function or block, add the following line of code only once after you have initialized the XADataSource object.

```
oracle.tmm.jta.TrmConfig.initXADataSource((XADataSource)xaDs);
```

XADataSource is an interface defined in JTA whose implementation is provided by the JDBC driver.

The MicroTx Distributed Transactions client library uses this object to connect to database to start XA transactions and perform various operations such as prepare, commit, and rollback. The MicroTx Distributed Transactions library also provides a SQL connection object to the application code to execute DML using dependency injection.

7. Insert the following line in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions client library.

```
@Inject
@MicroTxXATopicSession
private XATopicSession xaTopicSession;
```

Where, MicroTxXATopicSession is the name of that you have provided in `jms.topicName` in the `application.yaml` file.

8. When you inject MicroTxXATopicSession, MicroTx Distributed Transactions client library automatically creates a session object and injects a database connection object. So, you do not have to inject a connection object.

The following code snippet demonstrates how you can retrieve the details of a database connection object while using the `MicroTxXATopicSession` object.

```
connection = ((AQjmsSession) xaTopicSession).getDBConnection();
```

9. Use the `MicroTxXATopicSession` object, injected by the MicroTx Distributed Transactions client library, to publish messages to TEQ and for any DML operation on Oracle Database. Insert the following lines in the code of the participant service so that the service uses the injected connection object whenever the participant service performs a DML operation.

```
Statement stmt1 = connection.createStatement();  
stmt1.execute(query);  
stmt1.close();
```

Where, `connection` is the name of the `Connection` object that you have injected in the previous step.

Insert these lines of code for every DML operation that your participant service performs. Create a new statement object, such as `stmt1` or `stmt2` for every DML operation, but use the same connection object that is created by the MicroTx Distributed Transactions client library.

10. Save the changes.

Source code of a sample JAX-RS transaction participant application which uses the MicroTx Distributed Transactions XA library functions to leverage TEQ is available in the `xa/java/jms/department-helidon-teq` folder in the [microtx-samples](#) GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.12.3 Configure Spring REST Participant App to Leverage Transactional Event Queues

Use the information provided in this section to integrate Oracle MicroTx Distributed Transactions library with your Spring REST participant application to leverage the Oracle Transactional Event Queues (TEQ) feature.

You can also perform the following steps for a Spring REST application that initiates and participates in an XA transaction.

1. Open the `application.yaml` file in any text editor and provide property values for the MicroTx Distributed Transactions client library and the database details.
 - In addition to the existing data source configuration details, enter a unique TEQ topic name for the session.

```
jms:  
  topicName: "microTxXATopicSession"
```

Remember this name as you will use it later to inject a `TopicSession` object. If you are using multiple resource managers, enter these details in the `application.yaml` file for each resource manager.

- Provide details to connect to the Oracle Database where TEQ is available. The following example provides sample database URL, password, user name, and RMID

values for the departmentDataSource database. For description of the properties, see Data Store Properties.

```
departmentDataSource:
  url: "jdbc:oracle:thin:@tcps://<host>:<port>/<service_name>?"
  wallet_location=Database_Wallet"
  user: "dbuser"
  password: "xxxxxx"
  rmid: "174A5FF2-D8B1-47B0-AF09-DA5AFECA2F61"
  # Properties for using Universal Connection Pool (UCP)
  # Note: These properties require JDBC version 21.0.0.0
  oracleucp:
    driver-class-name: oracle.jdbc.OracleDriver
    type: oracle.ucp.jdbc.PoolXADataSource
    connection-factory-class-name:
oracle.jdbc.xa.client.OracleXADataSource
    sql-for-validate-connection: select * from dual
    connection-pool-name: connectionPoolName2
    initial-pool-size: 15
    min-pool-size: 10
    max-pool-size: 30
    data-source-name: deptxdatasource
```

Only if you use multiple resource managers, specify a unique resource manager ID (RMID) for each resource manager. Also specify one of these RMIDs in the MicroTx library properties. If you use a single resource manager, you don't need to mention the RMID in the application.yaml file.

- The following example provides sample values for the MicroTx Distributed Transactions client library properties. Provide the values based on your environment.

```
spring:
  microtx:
    participant-url: https://bookTicket-app:8081
    propagation-active: true
    http-client-connection-pool-size: 15
    xa-transaction-timeout: 60000
    xa-resource-manager-id: 174A5FF2-D8B1-47B0-AF09-DA5AFECA2F61
    xa-xa-support: true
```

Ensure that xa-xa-support is set to true.

For details about each property and other optional properties, see [Configure Library Properties for Spring Boot Apps](#).

2. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.

```
<dependency>
  <groupId>jakarta.jms</groupId>
  <artifactId>jakarta.jms-api</artifactId>
  <version>3.1.0</version>
</dependency>

<dependency>
```

```

        <groupId>com.oracle.database.messaging</groupId>
        <artifactId>aqapi-jakarta</artifactId>
        <version>26.1</version>
    </dependency>

```

3. Import the `jakarta.jms.XATopicSession` package.

```
import jakarta.jms.XATopicSession;
```

4. Initialize an `XADataSource` object.

The MicroTx Distributed Transactions client library needs to access an `XADataSource` object. It uses this object to create `XAConnection` and `XAResource` objects to connect with a resource manager or database server. The following code describes how you can define the `XADataSource` object at the beginning of the application code when you create the connection object.

```

class oracle.tmm.jta.MicroTxConfig
static void initXaDataSource(XADataSource xaDs)

```

For more information about `XADataSource`, see <https://docs.oracle.com/javase/8/docs/api/javax/sql/XADataSource.html>.

5. In the transaction participant function or block, specify the `XADataSource` object which is used by the MicroTx Distributed Transactions client library. Provide the credentials and other details to connect to the resource manager.

```

//Example for a participant using an Oracle Database:
OracleXADataSource dataSource = new
oracle.jdbc.xa.client.OracleXADataSource();
dataSource.setURL(url); //database connection string
dataSource.setUser(user); //username to access database
dataSource.setPassword(password); //password to access database
MicroTxConfig.initXaDataSource((XADataSource)dataSource);

```

It is the responsibility of the application developer to ensure that an XA-compliant JDBC driver and required parameters are set up while allocating `XADataSource`.

The MicroTx Distributed Transactions client library uses the `XADataSource` object to create database connections.

6. In the transaction participant function or block, add the following line of code only once after you have initialized the `XADataSource` object.

```
oracle.tmm.jta.MicroTxConfig.initXaDataSource((XADataSource)xaDs);
```

`XADataSource` is an interface defined in JTA whose implementation is provided by the JDBC driver.

The MicroTx Distributed Transactions client library uses this object to connect to database to start XA transactions and perform various operations such as prepare, commit, and rollback. The MicroTx Distributed Transactions library also provides a SQL connection object to the application code to execute DML using dependency injection.

7. Insert the following line in the code of the participant service so that the application uses the connection passed by the MicroTx Distributed Transactions client library. The following

code in the participant application injects the `microTxXATopicSession` object that is created by the MicroTx Distributed Transactions client library.

```
@Autowired
@Qualifier("microTxXATopicSession")
@Lazy
private XATopicSession xaTopicSession;
```

Where, `microTxXATopicSession` is the name of that you have provided in `jms.topicName` in the `application.yaml` file.

The MicroTx Distributed Transactions client library uses this object to connect to database to start XA transactions and perform various operations such as prepare, commit, and rollback. The MicroTx Distributed Transactions library also provides a SQL connection object to the application code to execute DML using dependency injection.

8. When you inject `MicroTxXATopicSession`, MicroTx Distributed Transactions client library automatically creates a session object and injects a database connection object. So, you do not have to inject a connection object.

The following code snippet demonstrates how you can retrieve the details of a connection object while using the `microTxXATopicSession` object.

```
connection = ((AQjmsSession) xaTopicSession).getDBConnection();
```

9. Use the object injected by the MicroTx Distributed Transactions client library to publish messages to TEQ and for any DML operation on Oracle Database. Insert the following lines in the code of the participant service so that the service uses the injected connection object whenever the participant service performs a DML operation.

```
Statement stmt1 = connection.createStatement();
stmt1.execute(query);
stmt1.close();
```

Where, `connection` is the name of the `Connection` object that you have injected in the previous step.

Insert these lines of code for every DML operation that your participant service performs. Create a new statement object, such as `stmt1` or `stmt2` for every DML operation, but use the same connection object that is created by the MicroTx Distributed Transactions client library.

10. Save the changes.

Source code of a sample Spring REST transaction participant application which uses the MicroTx Distributed Transactions XA library functions to leverage TEQ is available in the `xa/java/jms/departments-spring-teq` folder in the [microtx-samples](#) GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.13 Develop Node.js Apps with XA

- [Configure Node.js App as Transaction Initiator](#)

A transaction initiator service initiates or starts a transaction. Based on your application's business logic, a transaction initiator service may only start the transaction or start the transaction and participate in the transaction as well.

- [Configure Node.js App as Transaction Participant](#)
Depending on whether your resource manager is compliant with XA or not, set environment variables and implement different classes from the MicroTx Distributed Transactions client library.

3.13.1 Configure Node.js App as Transaction Initiator

A transaction initiator service initiates or starts a transaction. Based on your application's business logic, a transaction initiator service may only start the transaction or start the transaction and participate in the transaction as well.

Before you begin, identify if your application only initiates the transaction or initiates and participates in the transaction. Configure your application accordingly as the requirements vary slightly for the two scenarios.

Let us consider two scenarios to understand if your application only initiates the transaction or participates in the transaction as well.

- Scenario 1: A banking teller application transfers an amount from one department to another. Here, the teller application only initiates the transaction and does not participate in it. Based on the business logic, the teller application calls different services to complete the transaction. A database instance may or may not be attached to the teller application.
- Scenario 2: A banking teller application transfers an amount from one department to another. For every transaction, the teller application charges 1% as commission. Here, the teller application initiates the transaction and participates in it. A database instance must be attached to the teller application to save the transaction information.

To configure your Node.js application as a transaction initiator:

1. Add the MicroTx Distributed Transactions library for Node.js as a dependency in the package.json file.

```
"dependencies": {  
  "@oracle/microtx": "<version>"  
}
```

Where, <version> is the latest version of the MicroTx Distributed Transactions library for Node.js.

2. Configure the property values for the MicroTx Distributed Transactions library. See [Configure Library Properties for JAX-RS and Node.js Apps](#). To enable logging for Node.js applications, you must set additional properties. See [Enable Logs for MicroTx Distributed Transactions Node.js Library](#).
3. Configure the MicroTx Distributed Transactions library properties for the microservice by passing the tmm.properties file in which you have defined the values.

```
TrmConfig.init('./tmm.properties');
```

4. Edit the application code to:
 - a. Create a TrmUserTransaction object.
 - b. To begin a transaction, call begin() on the TrmUserTransaction object that you have created. The parameters that you pass when you call begin() depend on whether your application only initiates the transaction or also participates in it.
 - c. To commit or rollback the transaction, call commit() or rollback() on the TrmUserTransaction object that you have created.

The following example shows how to create a `TrmUserTransaction` object named `ut`, and then begin, commit or rollback a transaction. Here `req` represents the request.

```
//Step 3(a): Create a TrmUserTransaction object
let ut: TrmUserTransaction = newTrmUserTransaction();

//Optional. Specify the amount of time, in milliseconds, for which the
transaction remains active.
//If a transaction is not committed or rolled back within the specified
time period, the transaction is rolled back.
ut.setTimeout(60000);

try {
  //Step 3(b): Transaction demarcation - (start)
  await ut.begin(req); //If your application only initiates the
transaction and does not participate in it.
  await ut.begin(req, true); //If your application initiates the
transaction and participates in it.

  ... // implement business logic

  await ut.commit(req); //Step 3(c): Transaction demarcation - commit (end)

  resp.status(200).send("Transaction complete.");
}
catch (e) {
  console.log("Transaction Failed: ", e);
  let message = e.message;
  try {
    console.log("Rollback on transaction failure.");
    await ut.rollback(req); //Step 3.c: Transaction rollback (end)
    message = message + ". Transaction rolled back. ";
  } catch (ex) {
    console.log("Error in rollback for transfer failure: ", ex);
  }
  resp.status(500).send(message);
}
```

The example code is implemented in a try-catch statement, so that errors, if any, are handled gracefully. You can also implement your sample code without using a try-catch statement.

5. Save the changes, and then deploy your application. See [Deploy Your Application](#).

If the initiator service also participates in the transaction in addition to initiating the transaction, you must make additional configurations for the application to participate in the transaction and communicate with the resource manager. See [Configure Node.js App as Transaction Participant](#).

Source code of a sample Spring REST transaction initiator application which uses the MicroTx Distributed Transactions library and XA is available in the [teller](#) folder which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application. This provides an example of how you can use the MicroTx Distributed Transactions Spring REST libraries with the business logic of your Spring REST initiator application. This sample application is called `Teller`. It initiates a transaction between two departments. It calls Dept A to withdraw an amount and it calls Dept B to deposit the amount.

3.13.2 Configure Node.js App as Transaction Participant

Depending on whether your resource manager is compliant with XA or not, set environment variables and implement different classes from the MicroTx Distributed Transactions client library.

- [Configure Node.js Apps with an XA-Compliant Resource Manager](#)
Use the information provided in this section to configure your Node.js transaction participant applications when you use an XA-compliant resource manager.
- [Configure Node.js Apps with a Non-XA Resource](#)
Use the information provided in this section to configure your Node.js transaction participant applications when you use a non-XA resource, such as MongoDB.

3.13.2.1 Configure Node.js Apps with an XA-Compliant Resource Manager

Use the information provided in this section to configure your Node.js transaction participant applications when you use an XA-compliant resource manager.

1. Add the MicroTx Distributed Transactions library for Node.js as a dependency in the `package.json` file.

```
"dependencies": {  
  "@oracle/microtx": "<version>"  
}
```

Where, `<version>` is the latest version of the MicroTx Distributed Transactions library for Node.js.

2. Configure the property values for the MicroTx Distributed Transactions library. See [Configure Library Properties for JAX-RS and Node.js Apps](#). To enable logging for Node.js applications, you must set additional properties. See [Enable Logs for MicroTx Distributed Transactions Node.js Library](#).

Ensure that you set the value of `oracle.tmm.xa.XaSupport` as `true` and the value of `oracle.tmm.xa.LLRSupport` as `false`.

```
oracle.tmm.xa.XaSupport = true  
oracle.tmm.xa.LLRSupport = false
```

3. Configure the MicroTx Distributed Transactions library properties for the microservice by passing the `tmm.properties` file in which you have defined the values.

```
TrmConfig.init('./tmm.properties');
```

4. Import the MicroTx Distributed Transactions libraries.

```
import {Request, Response, Router} from 'express';  
import {XATransactionMethod, XAConfig, XADataSource, TrmXAResource} from  
"@oracle/microtx/xa/xa";  
import {TrmConfig} from "@oracle/microtx/util/trmutils";  
import {asyncHandler} from "@oracle/microtx/util/asynchandler";
```

5. If you are using Oracle Database as the resource manager, additionally import the following library.

```
import {OracleXADataSource} from "@oracle/microtx/xa/oraxa";
```

6. Create a router object.

For example, the following code creates a router object named `bankSvcRouter`. Provide a unique name for the router.

```
const bankSvcRouter = Router();
```

7. Use the following format to provide the database connection details in a parameter.

```
dbConfig = export default {
  user : "database_user",
  password : "database_password",
  connectionString : "database_connection_string"
};
```

Where,

- `dbConfig` is the name of the parameter that you want to create.
- `database_user` and `database_password` are the username and password to access the XA-compliant resource manager.
- `connectionString`: Enter the connection string to the data store in Oracle Database.
 - If you are using a non-autonomous Oracle Database (a database that does not use a credential wallet), use the following format to enter the connection string:

```
jdbc:oracle:thin:@<publicIP>:<portNumber>/<database unique
name>.<host domain name>
```

For example:

```
jdbc:oracle:thin:@123.213.85.123:1521/
CustDB_iad1vm.sub05031027070.customervcnwith.oraclevcn.com
```

- If you are using Oracle Database Cloud Service with Oracle Cloud Infrastructure, see [Create the Oracle Database Classic Cloud Service Connection String](#) in *Using Oracle Blockchain Platform*.
- If you are using Oracle Autonomous Transaction Processing, use the following format to enter the connection string:

```
jdbc:oracle:thin:@tcps://<host>:<port>/<service_name>?
wallet_location=<wallet_dir>
```

You can find the required details, such as host, port, and service name in the `tnsnames.ora` file, which is located in folder where you have extracted the wallet.

For example:

```
jdbc:oracle:thin:@tcps://adb.us-phoenix-1.oraclecloud.com:7777/
unique_connection_string_low.adb.oraclecloud.com?
wallet_location=Database_Wallet
```

8. Pass the parameter that contains the database connection details and create a `OracleXADataSource` object.

```
const xaPds: XADataSource = new OracleXADataSource(dbConfig);
```

9. Pass the `OracleXADataSource` object that you have created to the `TrmXAResource.init` method.

```
TrmXAResource.init(xaPds);
```

10. Call the `getXaConnection` method to initialize the database connection.

```
xaPds.getXaConnection();
```

11. Initialize `XAConfig` for all the REST API endpoints in the participant service that can participate in an XA transaction. There can be more than one endpoint methods that can participate in an XA transaction. Create an instance of `XATransactionMethod` for each endpoint, and then pass an array of `XATransactionMethod` into the `XAConfig` object.

The following code sample describes how you can initialize the objects for the `/deposit` end point.

```
const xaTransactionDeposit : XATransactionMethod = new
XATransactionMethod("/deposit");
const xaTransactionMethods : XATransactionMethod[] =
[xaTransactionDeposit];
const xaConfig: XAConfig = new XAConfig(bankSvcRouter, '/',
xaTransactionMethods);
```

12. This is setting up our interceptors in order to intercept calls to these endpoints with any current global transaction. The following code sample describes how the Express.js router, `bankSvcRouter`, routes incoming requests for the specified endpoint, `/deposit` to the functions you specify.

```
//This is an endpoint that can participate in an XA transaction.
bankSvcRouter.post('/deposit', (req, resp) => {
  doDeposit(req, resp); //business logic
});

async function doDeposit(req: Request, resp: Response) {
  console.log(`Nodejs department Service deposit() called`);
  //The following sample code demonstrates how you can use the connection
  object within your business logic.
  let amount = 10;
  if (req.query.amount != null && typeof req.query.amount === 'string') {
    amount = parseInt(req.query.amount, 10);
  }
  // XA connection pool is created and managed by the MicroTx
```

```

Distributed Transactions library
// and is present in the context property of req object.
// This is available on endpoints that are part of a XA transaction.
try {
    await req.context.xaConnection.connection.execute('UPDATE accounts
SET amount = amount + :1 where account_id = :2', [amount, req.params.id]);
    resp.status(200).send();
} catch (e: any) {
    resp.status(500).send();
}
}

```

Source code of a sample Spring REST transaction participant application which uses the MicroTx Distributed Transactions library and XA transaction protocol is available in the [department](#) folder which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.13.2.2 Configure Node.js Apps with a Non-XA Resource

Use the information provided in this section to configure your Node.js transaction participant applications when you use a non-XA resource, such as MongoDB.

1. Add the MicroTx Distributed Transactions library for Node.js as a dependency in the `package.json` file.

```

"dependencies": {
  "@oracle/microtx": "<version>"
}

```

Where, `<version>` is the latest version of the MicroTx Distributed Transactions library for Node.js.

2. Configure the property values for the MicroTx Distributed Transactions library. See [Configure Library Properties for JAX-RS and Node.js Apps](#). To enable logging for Node.js applications, you must set additional properties. See [Enable Logs for MicroTx Distributed Transactions Node.js Library](#).

Ensure that you set the value of `oracle.tmm.xa.XaSupport` as `false` and the value of `oracle.tmm.xa.LLRSupport` as `true`.

```

oracle.tmm.xa.XaSupport = false
oracle.tmm.xa.LLRSupport = true

```

3. Configure the MicroTx Distributed Transactions library properties for the microservice by passing the `tmm.properties` file in which you have defined the values.

```

TrmConfig.init('./tmm.properties');

```

4. Import the MicroTx Distributed Transactions libraries.

```

import {Request, Response, Router} from 'express';
import {XATransactionMethod, XAConfig, TrmConfig, NonXAResource,
TrmNonXAResource} from "../trmlib/xa";

```

5. Create a router object.

For example, the following code creates a router object named `bankSvcRouter`. Provide a unique name for the router.

```
const bankSvcRouter = Router();
```

6. Implement the `NonXAResource` interface.

For example, in the following code sample the `MongoDbNonXAResource` class implements the `NonXAResource` interface.

```
public class MongoDbNonXAResource implements NonXAResource {  
    // Provide application-specific code for all the methods in the  
    NonXAResource interface.  
}
```

7. Register the class, which implements the `NonXAResource` interface, with the `MicroTx Distributed Transactions` library for processing the XA operations.

The following example describes how you can register the `MongoDbNonXAResource` class, which implements the `NonXAResource` interface, with the `MicroTx Distributed Transactions` library.

```
const nonxaResource: NonXAResource = new MongoNonXAResource();
```

8. Use the `TrmNonXAResource.init()` function to specify the `NonXAResource` object that the `MicroTx Distributed Transactions` library uses.

```
TrmNonXAResource.init(nonxaResource)
```

9. Save the changes.

3.14 Develop ORDS App with XA

Configure an Oracle Database application, that you have built using Oracle APEX and Oracle REST Data Services (ORDS), as a transaction initiator or participant in an XA transaction with `MicroTx Distributed Transactions`.

A database application is an Oracle APEX and ORDS application which uses an Oracle Database. You can run the database application in a managed APEX service in Oracle Cloud Infrastructure, Oracle REST Data Services (ORDS) included in Oracle Autonomous Transaction Processing (ATP) and Autonomous Database (ADB), an Oracle RAD stack deployed in a Kubernetes cluster, or an Oracle RAD stack deployed within a VM or a physical host. The Oracle RAD stack is an inclusive technology stack based on three core components: Oracle REST Data Services (ORDS), Oracle APEX, and Oracle Database.

- [Prerequisites](#)
- [Run MicroTx Distributed Transactions PL/SQL Library](#)
The `MicroTx Distributed Transactions PL/SQL Library` for XA provides a set of functions and stored procedures for an Oracle Database application to participate in an XA transaction that is coordinated by `MicroTx Distributed Transactions`.
- [Configure PL/SQL Library Properties for ORDS Apps](#)
Provide configuration information for the `MicroTx Distributed Transactions PL/SQL Library` properties for ORDS application. The property values that you must provide would vary depending on whether the application only participates in the transaction or initiates it.

- [Privileges](#)
Set the privileges required to run the MicroTx Distributed Transactions PL/SQL library.
- [Configure ORDS App as Transaction Initiator](#)
- [Configure ORDS App as Transaction Participant](#)
- [MicroTx Distributed Transactions PL/SQL Library Functions](#)
The MicroTx Distributed Transactions PL/SQL library provides the following functions to specify the transaction boundaries.

3.14.1 Prerequisites

Before you begin, complete the following tasks.

- Create or identify a working stack comprising of Oracle REST Data Services (ORDS), Oracle APEX, and Oracle Database. This stack can run in the same Kubernetes cluster in which MicroTx Distributed Transactions runs or it can run in any other environment.
- Ensure that there is network access or connectivity between MicroTx Distributed Transactions and the database application if you have not deployed them in the same Kubernetes cluster.
- Use an existing schema or create a new schema in Oracle Database. Ensure that you register the schema with ORDS. See *How to Access RESTful Services in Oracle® Application Express SQL Workshop Guide*.
- Ensure that the ORDS service is available for the schema you have registered. For example, `http://localhost:50080/ords`.
- Log in to your SQL Developer Web interface using the admin or sysdba user credentials. To enable SQL Developer Web, see *Enabling User Access to SQL Developer Web in Using Oracle® SQL Developer Web*.
- Grant connect privileges to the APEX database user. See *Granting Connect Privileges in Oracle® APEX Installation Guide*.
- Add permissions by creating an access control list (ACL) if outbound REST calls are not allowed by default.

The MicroTx Distributed Transactions library makes an outbound REST call to the MicroTx Distributed Transactions transaction coordinator for enlisting the participant service into an XA transaction.

Create the required ACL, configure it, and then add it to the database. You will need sysdba permissions to add an ACL. The following example shows a sample ACL which is used only in restricted environments, such as production environments. For more information about adding an ACL, see the APEX documentation.

```
/
BEGIN
DBMS_NETWORK_ACL_ADMIN.APPEND_HOST_ACE(
  host => '#TMM_HOST_NAME',
  lower_port => null,
  upper_port => null,
  ace => xs$ace_type(privilege_list => xs$name_list('connect', 'resolve',
    'http'),
  principal_name => '#PRINCIPAL_NAME',
  principal_type => xs_acl.ptype_db));
```

```
END;  
/
```

Where, you must replace the following values with values that are specific to your environment.

- `#TMM_HOST_NAME`: Enter the host name or the external IP address of MicroTx Distributed Transactions.
- `#PRINCIPAL_NAME`: Enter the name of the principal user of APEX.

3.14.2 Run MicroTx Distributed Transactions PL/SQL Library

The MicroTx Distributed Transactions PL/SQL library for XA provides a set of functions and stored procedures for an Oracle Database application to participate in an XA transaction that is coordinated by MicroTx Distributed Transactions.

ORDS exposes REST APIs that are defined in Oracle Database using PL/SQL and APEX libraries. Integrate the MicroTx Distributed Transactions PL/SQL library file with your ORDS application. The library is available as a SQL file that you must run before executing the application code. You must perform this one-time task to install the library.

1. Connect to the Oracle Database using the schema user that you have registered with ORDS.

You can connect using SQL Developer or SQLPlus.

2. Run the `tmmxa.sql` file using SQL Developer or SQL Plus in an existing schema or create a new schema.

This file is located in the `installation_directory/otmm-RELEASE/lib/plsql` folder.

This creates a set of PL/SQL functions and stored procedures that are available only in the schema where you ran the SQL file. It also creates the following tables:

- `microtx_config`, which stores the configuration information for the MicroTx Distributed Transactions PL/SQL library properties. See [Configure PL/SQL Library Properties for ORDS Apps](#).
- `microtx_log_data`, which stores the logs of the operations performed by the MicroTx Distributed Transactions PL/SQL library.

The exceptions thrown by the MicroTx Distributed Transactions PL/SQL library are defined in the `tmm_exceptions` package.

3.14.3 Configure PL/SQL Library Properties for ORDS Apps

Provide configuration information for the MicroTx Distributed Transactions PL/SQL library properties for ORDS application. The property values that you must provide would vary depending on whether the application only participates in the transaction or initiates it.

Provide property values for the MicroTx Distributed Transactions PL/SQL library in the `microtx_config` table. The `microtx_config` table is created and populated with default values when you run the `tmmxa.sql` file.

- `coordinator-url`: Enter the URL to access the MicroTx Distributed Transactions coordinator. See [Access MicroTx Distributed Transactions](#). You must enter this value for the transaction initiator application. You don't have to specify this value for the transaction participant applications.

- `callback-url`: Enter the URL of your initiator or participant service. MicroTx Distributed Transactions uses the URL that you provide to connect to the service. Provide this value in the following format:

`https://externalHostnameOfApp:externalPortOfApp/`

Where,

- `externalHostnameOfApp`: The external host name of your initiator or participant service. For example, `bookTicket-app`.
- `externalPortOfApp`: The port number over which you can access your initiator or participant service remotely. For example, `8081`.

You must specify this value for the transaction participant applications.

For example, `https://bookTicket-app:8081`.

- `propagation-active`: Set this to true when you want to trace the transaction from end-to-end. This propagates the trace headers for all incoming and outgoing requests.
- `xa-transaction-timeout`: Specify the maximum amount of time, in milliseconds, for which the transaction remains active. If a transaction is not committed or rolled back within the specified time period, the transaction is rolled back. The default value and minimum value is 60000. Specify this value for both initiator and participant applications.
- `resource-manager-id`: Specify a unique string value for each resource manager that you use in the XA transaction. The unique value that you provide as RMID is used by MicroTx Distributed Transactions to identify the resource manager. If more than one participant uses the same resource manager, then specify the same resource manager ID for the participants that share a resource manager. This value is not related to any properties of the data store. For example, `174A5FF2-D8B2-47B0-AF09-DA5AFECA2F71`.
- `logging-enabled`: Set this to true to store the logs of the operations performed by the MicroTx Distributed Transactions PL/SQL library in the `microtx_log_data` table. Default value is true.
- `debug-logging-enabled`: Set this to true to set the log level to debug. Use this setting when you want to debug an issue as it logs all the events. The default value is false, which sets the log level to info and logs the least amount of information.

For example,

```
'coordinator-url' : 'http://tmm-app:9000/api/v1'
'callback-url'   : 'http://bookTicket-app:8081'
'resource-manager-id' : CONCAT(CONCAT(sys_context( 'userenv',
'current_schema' ), '-'), SYS_GUID()))'
'propagation-active' : 'true'
'xa-transaction-timeout' : '60000'
'logging-enabled' : 'true'
'debug-logging-enabled' : 'true'
```

3.14.4 Privileges

Set the privileges required to run the MicroTx Distributed Transactions PL/SQL library.

The following privileges are required to run the MicroTx Distributed Transactions PL/SQL library:

- CREATE SESSION
- CREATE TABLE
- CREATE PROCEDURE
- CREATE SEQUENCE
- CREATE TYPE

To perform XA operations using the MicroTx Distributed Transactions PL/SQL library, you must set the following privileges.

- GRANT SELECT ON DBA_PENDING_TRANSACTIONS TO user;
- GRANT FORCE ANY TRANSACTION TO user;

For more information, see DBMS_XA Data Structures in *PL/SQL Packages and Types Reference*.

The MicroTx Distributed Transactions PL/SQL library is available only in the schema where you ran the tmmxa.sql file. If you want to access the library components cross schema, then it is the database administrator's responsibility to provide the required permissions to run the library functions.

Run all library functions, procedures, types, packages with invoker rights or the privileges of the current user. Set AUTHID to indicate that you want to use invoker's rights. For example:

```
CREATE PROCEDURE proc1 AUTHID CURRENT_USER ...
```

For more information on AUTHID and privileges, see *Oracle Database PL/SQL Language Reference*.

3.14.5 Configure ORDS App as Transaction Initiator

1. Specify property values for the MicroTx library. See [Configure PL/SQL Library Properties for ORDS Apps](#).
2. For every REST API, define a template and a handler.
 - a. Enter a name and base path for the REST service module. The following code example provides teller and transfers as values. Replace these values with information that is specific to your environment.

```
DECLARE
    //Provide a name for the REST service module
    restModuleName VARCHAR2(256):= 'teller';
    //Provide a base path for the REST service
    restModuleBasePath VARCHAR2(256):= 'transfers';
```

- b. Define a module, and then define a template for the module.

```
BEGIN
    ORDS.define_module(
        p_module_name    => restModuleName,
        p_base_path      => restModuleBasePath,
        p_items_per_page => 0);
```

```
ORDS.define_template(
    p_module_name => restModuleName,
    p_pattern      => 'transfer');
```

- c. Provide source code for the module. The following code snippet demonstrates how you can define transaction boundaries by using the MicroTx Distributed Transactions PL/SQL library functions, `tmm_begin_tx()`, `tmm_rollback_tx()`, and `tmm_commit_tx()`. When you begin a transaction, it returns a `MicroTxTransaction` object. You will use this returned object to commit or roll back the global transaction. For reference information about the PL/SQL functions, see [MicroTx Distributed Transactions PL/SQL Library Functions](#).

```
ORDS.define_handler(
    p_module_name => restModuleName,
    p_pattern      => 'transfer',
    p_method       => 'POST',
    p_source_type  => ORDS.source_type_plsql,
    p_source       => '
    DECLARE
        payload          JSON_OBJECT_T;
        l_from_account   VARCHAR2(32);
        l_to_account     VARCHAR2(32);
        l_amount         VARCHAR2(32);
        l_microTxTransaction MicroTxTransaction;
        l_forwardHeaders ForwardHeaders;
        l_is_withdraw_successful BOOLEAN;
        l_is_deposit_successful BOOLEAN;
    BEGIN
        payload := JSON_OBJECT_T(:body);
        l_from_account := payload.get_String('from');
        l_to_account := payload.get_String('to');
        l_amount := payload.get_String('amount');

        -- Set response header
        OWA_UTIL.mime_header('application/json', FALSE);
        OWA_UTIL.http_header_close;

        -- set forward headers
        -- the bind variables are case sensitive
        l_forwardHeaders :=
ForwardHeaders(:authorization, :tmmTxToken, :requestId);

        -- tmm_begin_tx starts a global transaction that spans
        across multiple microservices.
        -- It returns a MicroTxTransaction object which has the
        transaction metadata.
        -- 60000 is the time, in milliseconds, for which the
        transaction remains active.
        -- If a transaction is not committed or rolled back within
        the specified time period, the transaction is rolled back.
        l_microTxTransaction := tmm_begin_tx(l_forwardHeaders,
60000);

        l_is_withdraw_successful :=
```

```

callWithdrawParticipant(l_from_account, l_amount,
l_microTxTransaction, l_forwardHeaders);
    IF NOT l_is_withdraw_successful THEN
        tmm_rollback_tx(l_microTxTransaction, l_forwardHeaders);
        :status_code := 500;
        HTP.p('{"error" : "Withdraw failed"}');
        RETURN;
    END IF;

    l_is_deposit_successful :=
callDepositParticipant(l_to_account, l_amount, l_microTxTransaction,
l_forwardHeaders);
    IF NOT l_is_deposit_successful THEN
        tmm_rollback_tx(l_microTxTransaction, l_forwardHeaders);
        :status_code := 500;
        HTP.p('{"error" : "Deposit failed"}');
        RETURN;
    END IF;

    -- Within global transaction
    creditFundTransferRewards(l_from_account, l_amount);

    tmm_commit_tx(l_microTxTransaction, l_forwardHeaders);
    :status_code := 200;
    HTP.P('{"message":"Transfer successful"}');
    END;',
    p_items_per_page => 0);

```

All the external calls and DML statements that an initiator application runs after calling `tmm_begin_tx()` and before calling `tmm_commit_tx()` is considered as part of a single global transaction. In case `tmm_rollback_tx()` is called, it rolls back all external calls and DML statements that are executed after the application calls `tmm_begin_tx()` and before the application calls `tmm_commit_tx()`.

3. You must create a link header for any external requests that the initiator application makes. The following example demonstrates how you can create a link header with the name `link`, and then call the `getTmmLinkHeader()` function.

```

-- Specify the name for the link header
p_name_01 => 'Link',
p_value_01 => getTmmLinkHeader(l_microTxTransaction),

```

Where, `l_microTxTransaction` is the `MicroTxTransaction` object that the `TMM_BEGIN_TX` function returns. This object contains the transaction metadata.

Source code of a sample ORDS transaction initiator application which uses the `MicroTx` Distributed Transactions PL/SQL library functions is available in the `ords_initiator_app.sql` file which is located in the `xa/plsql/databaseapp` folder in the [microtx-samples](#) GitHub repository. You can use this as a reference while integrating the `MicroTx` libraries with your application.

3.14.6 Configure ORDS App as Transaction Participant

1. Specify property values for the `MicroTx` library. See [Configure PL/SQL Library Properties for ORDS Apps](#).

2. Create DDLs for tables and other database objects required for your application.
3. Add required DMLs to insert default data.
4. Create or define a new REST module for your application.
5. Create the required PL/SQL functions and stored procedures for your application.
6. For every REST API, define a template and a handler.
 - a. Enter a name and base path for the REST service module. The following code example provides accounts as a value. Replace this value with information that is specific to your environment.

```
DECLARE
    //Provide a name for the REST service module
    restModuleName VARCHAR2(256):= 'accounts';
    //Provide a base path for the REST service
    restModuleBasePath VARCHAR2(256):= 'accounts';
```

- b. Set value for the `l_callbackUrl`. For example, `http://localhost:50080/ords/ordstest/accounts`. Also initialize parameters for `TmmReturn`.

```
DECLARE
    //Set up the callbackUrl correctly. This is generally the base URL or
    //path of the module.
    l_callbackUrl VARCHAR2(256) := OWA_UTIL.get_cgi_env('X-APEX-BASE')
    || 'accounts';
    l_tmmReturn TmmReturn;
    l_tmmReturn2 TmmReturn;
```

- c. Call `TmmStart`.
The following code sample demonstrates how you can call the `TmmStart` function. Pass all the parameters as shown in the following example. You must pass the value for the `l_callbackUrl` when you call `TmmStart`. The values of all the other parameters are automatically obtained from the incoming request headers and passed.

```
//Call TmmStart. Specify value for callbackUrl.
l_tmmReturn := TmmStart(callbackUrl => l_callbackUrl, linkUrl
=> :linkUrl, requestId => :requestId, authorizationToken
=> :authorization, tmmTxToken => :tmmTxToken);
```

The `TmmStart` function returns an object, which provides an attribute, `proceed`, that indicates whether the `TmmStart` function was successfully executed and that the transaction can proceed ahead.

- d. Check if the XA transaction should proceed further (value of `l_tmmReturn.proceed` is greater than 0) or not (value of `l_tmmReturn.proceed` is 0). Execute your business logic only if the XA transaction can proceed further, otherwise the `TmmStart` function must return a HTTP error status code as shown in the following code sample. Call the `TmmEnd` function after the business logic has been completely executed.

```
IF (l_tmmReturn.proceed > 0) THEN
    //Execute your business logic only if the XA transaction can proceed
    //further.
    //Execute SQLs statements or call other functions or stored procedures.
    doWithdraw(p_amount => :amount, p_account_id => :accountId);
```

```

        //Call TmmEnd at the end of the REST function.
        l_tmmReturn2 := TmmEnd(p_xid => l_tmmReturn.xid);
        :status_code := 200;
    ELSE
        :status_code := 400; --bad request
    END IF;

```

- e. Create MicroTx Distributed Transactions callback APIs.

```
createTMMCallbacks(moduleName => restModuleName);
```

- f. Register all the method handlers that will participate in the XA transaction.

```

registerXaHandler(moduleName => restModuleName,
                  handlerPattern => ':accountId/withdraw',
                  handlerMethod => 'POST');

```

The following code sample demonstrates how you can implement the handler.

```

DECLARE
    //Provide a name for the REST service module
    restModuleName VARCHAR2(256) := 'accounts';
    //Provide a base path for the REST service
    restModuleBasePath VARCHAR2(256) := 'accounts';

BEGIN
    ORDS.define_module(
        p_module_name => restModuleName,
        p_base_path   => restModuleBasePath,
        p_items_per_page => 0);

    ORDS.define_template(
        p_module_name => restModuleName,
        p_pattern      => ':accountId/withdraw');

    ORDS.define_handler(
        p_module_name => restModuleName,
        p_pattern      => ':accountId/withdraw',
        p_method       => 'POST',
        p_source_type  => ORDS.source_type_plsql,
        p_source       => '
        DECLARE
            //Set up the callBackUrl correctly. This is
generally the base URL or path of the module.
            //Example: http://localhost:50080/ords/ordstest/
accounts
            l_callBackUrl VARCHAR2(256) :=
OWA_UTIL.get_cgi_env('X-APEX-BASE') || 'accounts';
            l_tmmReturn TmmReturn;
            l_tmmReturn2 TmmReturn;

            BEGIN
                //Call TmmStart. Pass all the other parameters
than the callBackUrl.
                l_tmmReturn := TmmStart(callBackUrl =>
l_callBackUrl, linkUrl => :linkUrl, requestId => :requestId,

```

```

authorizationToken => :authorization, tmmTxToken => :tmmTxToken);

                                //Check if the transaction should proceed
further                                // (value of l_tmmReturn.proceed is greater
                                //or not (value of l_tmmReturn.proceed is 0).
than 0)                                //Execute your business logic only if
                                transaction can proceed further.
                                //If not, then return with an HTTP error code.
                                IF (l_tmmReturn.proceed > 0)
THEN
                                //Execute your business logic.
                                //Execute SQLs statements or call other
functions or stored procedures.
                                doWithdraw(p_amount => :amount,
p_account_id => :accountId);

                                //Call TmmEnd at the end of the REST
function.
                                l_tmmReturn2 := TmmEnd(p_xid =>
l_tmmReturn.xid);

                                :status_code := 200;

                                ELSE
                                :status_code := 400; --bad request

                                END IF;

                                exception
                                when others then
                                :status_code := 500;

                                END;',
p_items_per_page => 0);

//Create MicroTx Distributed Transactions callback APIs.
createTMMCallbacks(moduleName => restModuleName);

//Register all method handlers that will participate in the XA
transaction.
registerXaHandler(moduleName => restModuleName,
handlerPattern => ':accountId/withdraw',
handlerMethod => 'POST');

COMMIT;
END;
/

```

Source code of an ORDS transaction participant application which uses the MicroTx Distributed Transactions PL/SQL library functions is available in `xa/plsql/databaseapp/ords_participant_app.sql` in the [microtx-samples](#) GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

3.14.7 MicroTx Distributed Transactions PL/SQL Library Functions

The MicroTx Distributed Transactions PL/SQL library provides the following functions to specify the transaction boundaries.

- [TMM_BEGIN_TX Function](#)
Used by a transaction initiator application to start a global transaction that spans across multiple microservices. It returns a MicroTxTransaction object which has the transaction metadata.
- [TMM_GET_TX_STATUS Function](#)
Used by a transaction initiator application to retrieve the current transaction status.
- [TMM_COMMIT_TX Function](#)
Used by the transaction initiator application to commit a global transaction.
- [TMM_ROLLBACK_TX Function](#)
Used by a transaction initiator application to roll back a global transaction.
- [TmmStart Function](#)
Enlists the transaction participant service with the MicroTx Distributed Transactions XA coordinator in a global transaction, starts the XA transaction boundary, registers the call back REST APIs, and returns the transaction metadata.
- [TmmEnd Function](#)
Call the TmmEnd function in your participant application code after the business logic has been completely executed. It ends the XA transaction boundary.

3.14.7.1 TMM_BEGIN_TX Function

Used by a transaction initiator application to start a global transaction that spans across multiple microservices. It returns a MicroTxTransaction object which has the transaction metadata.

Syntax

```
TMM_BEGIN_TX (  
    forwardHeaders      IN ForwardHeaders DEFAULT NULL,  
    transaction_timeout IN NUMBER)  
RETURN MicroTxTransaction;
```

TMM_BEGIN_TX Function Parameters

Parameter	Description
forwardHeaders	Optional. The application forwards the specified headers to the MicroTx Distributed Transactions coordinator, and then the coordinator uses these headers while communicating with the application. An application can forward the following headers: authorization token, transaction token, and a unique ID to trace the transaction across the microservices. If you have enabled authorization by setting <code>tmmConfiguration.authorization.authTokenPropagationEnabled</code> to true in the <code>values.yaml</code> file for the coordinator, your application must forward the authorization token. If you have set <code>tmmConfiguration.transactionToken.transactionTokenEnabled</code> to true in the <code>values.yaml</code> file for the coordinator, your application must forward the transaction token.
transaction_timeout	Optional. Specify the amount of time, in milliseconds, for which the transaction remains active. If a transaction is not committed or rolled back within the specified time period, the transaction is rolled back. The default value and minimum value is 60000. This overrides the value of <code>xa-transaction-timeout</code> in the <code>microtx_config</code> table.

Return Values

Parameter	Description
MicroTxTransaction	Returns the transaction object <code>MicroTxTransaction</code> . This object is required to join, commit, roll back, and get the status of the transaction.

Examples

The following sample code demonstrates how you can pass the forward headers to the `tmm_begin_tx` function and the function returns a `MicroTxTransaction` object.

```
//Set the headers that you want the application to forward to MicroTx
l_forwardheaders := ForwardHeaders(:authorization, :tmmTxToken, :requestId);
//Call the TMM_BEGIN_TX function to initiate the transaction
l_microTxTransaction := TMM_BEGIN_TX(l_forwardheaders);
```

Note down the name of the returned value as you will provide this later to get the transaction status, commit or roll back the transaction.

3.14.7.2 TMM_GET_TX_STATUS Function

Used by a transaction initiator application to retrieve the current transaction status.

Syntax

```
TMM_GET_TX_STATUS (
    microTxTransaction IN MicroTxTransaction,
    forwardHeaders      IN ForwardHeaders DEFAULT NULL)
RETURN VARCHAR2(64);
```

TMM_GET_TX_STATUS Function Parameters

Parameter	Description
microTxTransaction	Mandatory parameter. The MicroTxTransaction object that the TMM_BEGIN_TX function returns.
forwardHeaders	Optional. The application forwards the specified headers to the MicroTx Distributed Transactions coordinator, and then the coordinator uses these headers while communicating with the application. An application can forward the following headers: authorization token, transaction token, and a unique ID to trace the transaction across the microservices. If you have enabled authorization by setting <code>tmmConfiguration.authorization.authTokenPropagationEnabled</code> to true in the <code>values.yaml</code> file for the coordinator, your application must forward the authorization token. If you have set <code>tmmConfiguration.transactionToken.transactionTokenEnabled</code> to true in the <code>values.yaml</code> file for the coordinator, your application must forward the transaction token.

Return Values

Returns a VARCHAR2(64) value, which contains the status of the transaction for the specified MicroTxTransaction object.

Examples

The following sample code demonstrates how you can get the current transaction status using the TMM_GET_TX_STATUS function.

```
tx_status = TMM_GET_TX_STATUS (l_microTxTransaction, l_forwardheaders);
```

Where, `l_microTxTransaction` is the name of the MicroTxTransaction object that the TMM_BEGIN_TX function returns and `l_forwardheaders` that the headers that the application forwards.

3.14.7.3 TMM_COMMIT_TX Function

Used by the transaction initiator application to commit a global transaction.

Syntax

```
TMM_COMMIT_TX (  
    microTxTransaction IN MicroTxTransaction,  
    forwardHeaders     IN ForwardHeaders DEFAULT NULL  
);
```

TMM_COMMIT_TX Function Parameters

Parameter	Description
microTxTransaction	Mandatory parameter. The MicroTxTransaction object that the TMM_BEGIN_TX function returns.

Parameter	Description
forwardHeaders	Optional. The application forwards the specified headers to the MicroTx Distributed Transactions coordinator, and then the coordinator uses these headers while communicating with the application. An application can forward the following headers: authorization token, transaction token, and a unique ID to trace the transaction across the microservices. If you have enabled authorization by setting <code>tmmConfiguration.authorization.authTokenPropagationEnabled</code> to true in the <code>values.yaml</code> file for the coordinator, your application must forward the authorization token. If you have set <code>tmmConfiguration.transactionToken.transactionTokenEnabled</code> to true in the <code>values.yaml</code> file for the coordinator, your application must forward the transaction token.

Examples

The following sample code demonstrates how you can commit a transaction using the `TMM_COMMIT_TX` function.

```
TMM_COMMIT_TX (l_microTxTransaction, l_forwardheaders);
```

Where, `l_microTxTransaction` is the name of the `MicroTxTransaction` object that the `TMM_BEGIN_TX` function returns and `l_forwardheaders` that the headers that the application forwards.

3.14.7.4 TMM_ROLLBACK_TX Function

Used by a transaction initiator application to roll back a global transaction.

Syntax

```
TMM_ROLLBACK_TX (
    microTxTransaction IN MicroTxTransaction,
    forwardHeaders     IN ForwardHeaders DEFAULT NULL
);
```

TMM_ROLLBACK_TX Function Parameters

Parameter	Description
microTxTransaction	Mandatory parameter. The <code>MicroTxTransaction</code> object that the <code>TMM_BEGIN_TX</code> function returns.

Parameter	Description
forwardHeaders	Optional. The application forwards the specified headers to the MicroTx Distributed Transactions coordinator, and then the coordinator uses these headers while communicating with the application. An application can forward the following headers: authorization token, transaction token, and a unique ID to trace the transaction across the microservices. If you have enabled authorization by setting <code>tmmConfiguration.authorization.authTokenPropagationEnabled</code> to true in the <code>values.yaml</code> file for the coordinator, your application must forward the authorization token. If you have set <code>tmmConfiguration.transactionToken.transactionTokenEnabled</code> to true in the <code>values.yaml</code> file for the coordinator, your application must forward the transaction token.

Examples

The following sample code demonstrates how you can roll back a transaction using the `TMM_ROLLBACK_TX` function.

```
TMM_ROLLBACK_TX (l_microTxTransaction, l_forwardheaders);
```

Where, `l_microTxTransaction` is the name of the `MicroTxTransaction` object that the `TMM_BEGIN_TX` function returns and `l_forwardheaders` that the headers that the application forwards.

3.14.7.5 TmmStart Function

Enlists the transaction participant service with the MicroTx Distributed Transactions XA coordinator in a global transaction, starts the XA transaction boundary, registers the call back REST APIs, and returns the transaction metadata.

Syntax

```
TmmStart (
    callBackUrl      IN VARCHAR2,
    linkUrl          IN VARCHAR2,
    requestId        IN VARCHAR2,
    authorizationToken IN VARCHAR2,
    tmmTxToken       IN VARCHAR2)
RETURN TmmReturn;
```

TmmStart Function Parameters

Parameter	Description
callBackUrl	Mandatory. Enter the URL of your participant service and suffix it with the name of the ORDS REST module. MicroTx uses the URL that you provide to connect to the participant ORDS service. For example, <code>http://tmm-app:8080/ords/otmm/accounts</code> . Where, <code>/accounts</code> is the ORDS REST module endpoint.
linkUrl	Mandatory.
requestId	Mandatory. A unique ID to trace a transaction across microservices.

Parameter	Description
authorizationToken	Mandatory. If you have enabled authorization by setting <code>tmmConfiguration.authorization.authTokenPropagationEnabled</code> to <code>true</code> in the <code>values.yaml</code> file for the coordinator, your application must forward the authorization token.
tmmTxToken	Mandatory. If you have set <code>tmmConfiguration.transactionToken.transactionTokenEnabled</code> to <code>true</code> in the <code>values.yaml</code> file for the coordinator, your application must forward the transaction token.

Return Value

The `TmmStart` function returns a `TmmReturn` object, which has the transaction metadata. The `TmmReturn` object has an attribute, `proceed`, which indicates whether the `TmmStart` function was successfully executed and whether the transaction can progress.

TmmReturn.proceed Indicates that...

0	the <code>TmmStart</code> function was called within an XA transaction, but the XA initialization was not successful. So the application code must not proceed with the XA transaction.
1	the <code>TmmStart</code> function was called within an XA transaction and the XA initialization was successful. So the application code must proceed with the XA transaction.
2	there is no MicroTx Distributed Transactions XA transaction and the function has been executed within a local transaction. So the application code should proceed as normal.

Examples

The following sample code snippet demonstrates how you can call the `TmmStart` function and pass values to it. It returns a `MicroTxTransaction` object.

```
..
//In the example callback URL below, account is the name of the ORDS REST
module.
l_callbackUrl VARCHAR2(256) := get_microtx_config('callback-url') || '/'
accounts';
//Call the TmmStart function to initiate the transaction. Pass the
parameters, other than callbackUrl, as shown in the example below.
l_microTxTransaction := TmmStart(callbackUrl => l_callbackUrl, linkUrl
=> :linkUrl, requestId => :requestId, authorizationToken => :authorization,
tmmTxToken => :tmmTxToken);
//Check if the transaction should proceed or not
IF (l_microTxTransaction.proceed > 0) THEN
//Run the application's business logic
...
END IF;
```

Note down the name of the returned value as you will provide this later to get the transaction status, commit or roll back the transaction.

3.14.7.6 TmmEnd Function

Call the TmmEnd function in your participant application code after the business logic has been completely executed. It ends the XA transaction boundary.

Syntax

```
TmmEnd (  
    xid    IN DBMS_XA_XID)  
RETURN TmmReturn;
```

TmmEnd Function Parameters

Parameter	Description
xid	Unique ID for the transaction. When you call TmmStart(), it returns a TmmReturn object. You don't have to set a value for this parameter as this unique value is already set when the TmmReturn object is returned.

Return Value

The TmmEnd function returns a TmmReturn object, which has the transaction metadata.

Examples

The following sample code demonstrates how you can call the TmmEnd function and the function returns a TmmReturn object.

```
//Call the TmmEnd function to end the XA transaction boundary  
l_tmmReturn_end := TmmEnd(xid => l_tmmReturn.xid);
```

3.15 Include Oracle Blockchain Platform in XA Transactions

You can use MicroTx to manage XA transactions in Oracle Blockchain Platform for Java applications, such as Spring Boot and Helidon.

These applications may need to make updates across the Oracle Blockchain Platform ledger and other repositories such as databases or other blockchain ledgers in an atomic fashion, where either all updates succeed or none do. See [Include Oracle Blockchain Platform in Global Distributed Transactions](#) in *Using Oracle Blockchain Platform*.

4

Develop Applications with Saga

The Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions) library for Spring REST, Micronaut, and Node.js apps provides the functionality to initiate a new Saga transaction or to participate in an existing Saga transaction.

This section provides instructions to integrate MicroTx Distributed Transactions library with your application. Refer to the sample code of Travel Agent application that uses the Saga transaction protocol to book a trip. The MicroTx Distributed Transactions library files are already integrated with the application code. Link to the relevant files is available in each section. You can refer to the code of the sample application while you integrate the MicroTx Distributed Transactions libraries with your application.

Topics

- [Workflow to Develop Applications with Saga](#)
- [Propagate Oracle-Tmm-Tx-Token for multiple REST API calls](#)
When the business logic of your application spawns across multiple API calls to complete a transaction, you must propagate Oracle-Tmm-Tx-Token for multiple REST API calls.
- [Develop Java Apps with Saga](#)
You can develop JAX-RS and Spring-REST based applications that use the Saga transaction protocol.
- [Develop Node.js Apps with Saga](#)
The MicroTx Distributed Transactions library for Node.js provides the functionality to initiate a new Saga transaction or to participate in an existing Saga transaction. You must integrate this library into your Node.js application code.

4.1 Workflow to Develop Applications with Saga

Develop, test, and deploy your microservices independently. To use MicroTx Distributed Transactions to manage the transactions in your application, you need to make a few changes to your existing application code to integrate the functionality provided by the MicroTx Distributed Transactions libraries.

Use the following workflow as a guide to develop your applications to use MicroTx Distributed Transactions to manage Saga transactions.

Task	Description	More Information
Install MicroTx Distributed Transactions	Install MicroTx Distributed Transactions and ensure that you can access it.	Workflow to Install MicroTx
Provide configuration information for the MicroTx Distributed Transactions library properties.	Perform this step for all the transaction participant and transaction initiator Node.js applications, so that your Node.js applications can access the library.	Configure Library Properties for Node.js Apps

Task	Description	More Information
Integrate MicroTx Distributed Transactions library with your application code.	Select a suitable procedure to integrate the library based on the following factors: - the development framework for your application - whether an application initiates the transaction or participates in the transaction	The library is available for Java and Node.js apps. Perform one of the following tasks: Develop Java Apps with Saga Develop Node.js Apps with Saga
Enable session affinity	When you use internal memory as data store and deploy the transaction coordinator on more than one replica, then you must enable session affinity for Saga and XA transactions. You don't need to enable session affinity for TCC transactions.	Enable Session Affinity
Deploy your application	After integrating the MicroTx Distributed Transactions library files with your application, deploy the application.	Deploy Your Application

4.2 Propagate `Oracle-Tmm-Tx-Token` for multiple REST API calls

When the business logic of your application spawns across multiple API calls to complete a transaction, you must propagate `Oracle-Tmm-Tx-Token` for multiple REST API calls.

This section explains how the `Oracle_Tmm_Tx-Token` token is propagated when the business logic of your application spawns across multiple API calls to complete a transaction. If you have set `transactionTokenEnabled` to true in the YAML file and the business logic of your application spawns across multiple API calls to complete a transaction, you must retrieve the value of `Oracle_Tmm_Tx-Token` and pass it in the request header for all the subsequent API calls that the user makes.

Skip these steps if the business logic of your application requires only a single API call from a user to complete the entire transaction. To understand how the `Oracle_Tmm_Tx-Token` token is propagated when the business logic of your application requires only a single API call from a user, see [About the Oracle_Tmm_Tx-Token Transaction Token](#).

Let's consider a trip booking application, which requires two calls from a user. The first call is to initiate a transaction and make a provisional booking. The application requires a second API call from the user to confirm or cancel the booking. In such scenarios, when your application's business logic spawns across multiple API calls from a user to complete a single transaction, you must include `Oracle_Tmm_Tx-Token` in the request header of the subsequent API call from the user to confirm or cancel the booking.

The following steps describe how MicroTx Distributed Transactions creates the `Oracle_Tmm_Tx-Token` transaction token and propagates it for the first call and how you need to include `Oracle_Tmm_Tx-Token` in the subsequent API calls from a user.

1. When a user begins a transaction, the transaction initiator service sends a request to MicroTx Distributed Transactions.
2. MicroTx Distributed Transactions responds to the transaction initiator and returns `Oracle_Tmm_Tx-Token` in the response header.
The MicroTx Distributed Transactions library creates this token based on the private-public key pair that you provide. You don't have to create the `Oracle_Tmm_Tx-Token` transaction token or pass it in the request header.

MicroTx Distributed Transactions works with multiple headers and token. For the sake of simplicity, we are limiting our discussion to the `Oracle_Tmm_Tx-Token` transaction token in this section.

3. To secure calls from the participant services to the transaction coordinator, the MicroTx Distributed Transactions library passes `Oracle_Tmm_Tx-Token` in the request header for all the subsequent calls.
4. MicroTx Distributed Transactions returns `Oracle_Tmm_Tx-Token` in the response header while responding to the first call from the user. Retrieve the value of `Oracle_Tmm_Tx-Token` from the response header.
5. In all the subsequent API calls that the user makes, you must manually include the `Oracle_Tmm_Tx-Token` in the request header. Provide the value that you have retrieved in the previous step.

This ensures that the multiple API calls from a user are linked together and all the calls are considered as part of a single transaction.

4.3 Develop Java Apps with Saga

You can develop JAX-RS and Spring-REST based applications that use the Saga transaction protocol.

Eclipse Microprofile provides the annotations and APIs to coordinate Saga transactions for JAX-RS based REST applications. See <https://download.eclipse.org/microprofile/microprofile-lra-1.0-M1/microprofile-lra-spec.html#introduction>. For more information about the Saga annotations, see <https://download.eclipse.org/microprofile/microprofile-lra-1.0-M1/microprofile-lra-spec.html>.

Spring REST-based applications use the MicroTx Distributed Transactions Saga library for Spring Boot. JAX-RS and Spring REST-based applications use Saga annotations with the same names and function. However, the JAX-RS applications use the `import static org.eclipse.microprofile.lra.annotation.ws.rs.LRA` package, which is an Eclipse MicroProfile package. Spring REST-based applications uses the `com.oracle.microtx.springboot.lra.annotation.*` package, a custom MicroTx Distributed Transactions library file.

- [Develop Helidon Applications with Saga](#)
Helidon provides implementation for the Saga client specifications.
- [Configure MicroTx Distributed Transactions Library Properties for Java Apps](#)
Provide configuration information for the MicroTx Distributed Transactions library properties. You must perform this step for all Micronaut and Spring REST-based apps which participate or initiate the transaction and use the MicroTx Distributed Transactions library.
- [Develop Spring Boot Applications with Saga](#)
For Spring Boot applications that use the Saga transaction protocol, you must use the Java library file provided by MicroTx Distributed Transactions.
- [Develop Micronaut Apps with Saga](#)
You can integrate the MicroTx Distributed Transactions library files with Micronaut applications that use the Saga transaction protocol.
- [Develop Java Apps that Use Saga and Lock-Free Reservation](#)
MicroTx Distributed Transactions Saga libraries for Spring REST and Micronaut applications support lock-free reservation.

4.3.1 Develop Helidon Applications with Saga

Helidon provides implementation for the Saga client specifications.

For more information, see https://helidon.io/docs/v2/#/mp/lra/01_introduction. For Helidon applications that use the Saga transaction protocol, don't use the MicroTx Distributed Transactions library files. For information about the implementation for these applications, see <https://danielkec.github.io/blog/helidon/lra/saga/2021/10/12/helidon-lra.html>.

Refer to the source code of a sample Helidon transaction participant application which uses MicroTx Distributed Transactions to manage the Saga transaction. The files are available in the [hotel](#) folder in the `microtx-samples` GitHub repository.

4.3.2 Configure MicroTx Distributed Transactions Library Properties for Java Apps

Provide configuration information for the MicroTx Distributed Transactions library properties. You must perform this step for all Micronaut and Spring REST-based apps which participate or initiate the transaction and use the MicroTx Distributed Transactions library.

Do not perform this step for JAX-RS or Helidon apps as these apps do not use the MicroTx Distributed Transactions library.

Configure the following properties in the `application.properties` file of your Micronaut or Spring REST-based application. This enables the custom library to establish communication with the MicroTx Distributed Transactions Saga coordinator, participate in Saga transactions, and propagates the relevant headers for the coordinated transactions.

- `microtx.lra.coordinator-url` for Micronaut apps or `spring.microtx.coordinator-url` for Spring REST-based apps: Enter the URL to access the MicroTx Distributed Transactions Saga coordinator. This permits the MicroTx Distributed Transactions library to communicate and coordinate transactions with the coordinator. To get the URL, append `/lra-coordinator` to the URL to access MicroTx Distributed Transactions. For example, `https://192.0.2.1:443/api/v1/lra-coordinator`. For information to identify the URL to access MicroTx Distributed Transactions, see [Access MicroTx Distributed Transactions](#). You must enter this value for the transaction initiator application. You don't have to specify this value for the transaction participant applications.
- `microtx.lra.propagation-active` for Micronaut apps or `spring.microtx.lra.propagation-active` for Spring REST apps: Set this to true when you want to trace the transaction from end-to-end. This propagates the Saga context headers through non-LRA endpoints for all incoming and outgoing requests. This ensures proper coordination with other Saga participants.
- `microtx.lra.participant-url` for Micronaut apps or `spring.microtx.lra.participant-url` for Spring REST apps: Enter the URL of your participant service associated with the MicroTx Distributed Transactions client library. MicroTx Distributed Transactions uses the URL that you provide to connect to the participant service. Provide this value in the following format:

`https://externalHostnameOfApp:externalPortOfApp/`

Where,

- `externalHostnameOfApp`: The external host name of your initiator or participant service. For example, `bookTicket-app`.

- *externalPortOfApp*: The port number over which you can access your initiator or participant service remotely. For example, 8081.

You must specify this value for the transaction participant applications.

- *microtx.lra.headers-propagation-prefix* for Micronaut apps or *spring.microtx.lra.headers-propagation-prefix* for Spring REST apps: Specify a list of prefixes that you want to propagate in the header from the incoming requests to the outgoing requests within the custom library. For example, trace headers, authorization headers, and custom MicroTx Distributed Transactions headers. This ensures that necessary information is preserved during transactions as the specified prefixes are included as headers in the context of the propagated Saga transaction.
- *microtx.lra.lock-free-reservation-active* for Micronaut apps or *spring.microtx.lra.lock-free-reservation-active* for Spring-REST based apps: Set this to true to enable the MicroTx Distributed Transactions coordinator use the lock-free reservation feature provided in Oracle Database 23ai. When you set this value to true, MicroTx Distributed Transactions manages the lock-free reservation interaction and also injects and propagates the Long-Running-Action-SagaId header across the participants and coordinator calls. Do not provide a value for this property if your application does not use lock-free reservation.
- *spring.microtx.lra.retry-max-attempts* for Spring REST apps: Specify the maximum number of times that the application retries sending a request to the MicroTx Distributed Transactions coordinator in case of any failures. For example, 15. The default value is 10.
- *spring.microtx.lra.retry-max-delay* for Spring REST apps: The maximum retry interval, in milliseconds, before which the application retries sending the same request again to the MicroTx Distributed Transactions coordinator in case of any failures. For example, 2000. The default value is 1000 milliseconds.

Sample property values for Spring REST-based apps

```
spring.microtx.lra.coordinator-url=http://tmm-app:9000/api/v1/lra-coordinator
spring.microtx.lra.propagation-active=true
spring.microtx.lra.participant-url=http://bookTicket-app:8081
spring.microtx.lra.headers-propagation-prefix = {x-b3-, oracle-tmm-,
authorization, refresh-}
spring.microtx.lra.retry-max-attempts=15
spring.microtx.lra.retry-max-delay=2000
```

Sample property values for Micronaut apps

```
microtx.lra.coordinator-url=http://tmm-app:9000/api/v1/lra-coordinator
microtx.lra.propagation-active=true
microtx.lra.participant-url=http://bookTicket-app:8081
microtx.lra.headers-propagation-prefix = {x-b3-, oracle-tmm-, authorization,
refresh-}
```

You can use the HTTP protocol if your application and MicroTx Distributed Transactions are in the same Kubernetes cluster, otherwise use the HTTPS protocol.

4.3.3 Develop Spring Boot Applications with Saga

For Spring Boot applications that use the Saga transaction protocol, you must use the Java library file provided by MicroTx Distributed Transactions.

For reference information about the MicroTx Distributed Transactions library for Spring Boot applications, see [Oracle® Transaction Manager for Microservices Spring Boot API Reference for Saga](#).

To include the MicroTx Distributed Transactions library files and property values for Spring Boot applications:

1. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the Spring Boot application's pom.xml file. The following sample code is for the 23.4.1 release. Provide the correct version, based on the release version that you want to use.

Spring Boot 3.x applications work with Java 17 in Jakarta EE9 environments.

```
<dependency>
  <groupId>com.oracle.microtx.lra</groupId>
  <artifactId>microtx-lra-spring-boot-starter</artifactId>
  <version>23.4.1</version>
</dependency>
```

2. Specify values for the MicroTx Distributed Transactions library properties in the application.properties file of your application. This enables the custom library to establish communication with the MicroTx Distributed Transactions Saga coordinator, participate in Saga transactions, and propagate the relevant headers for the coordinated transactions. See [Configure MicroTx Distributed Transactions Library Properties for Java Apps](#).

3. Import the com.oracle.microtx.springboot.lra.annotation.* package.

```
import com.oracle.microtx.springboot.lra.annotation.*
```

4. Create an object, of a class in the MicroTx library, to call the endpoints of other Saga transaction participant services. The transaction initiator service begins the transaction. To complete the transaction, the initiator service may have to make calls to one or more participant services. While calling the participant services, use the object that you create.

-
- [RestTemplate](#)
 - [OpenFeign](#)

RestTemplate

Autowire the Spring Boot RestTemplate, provided by the MicroTx library.

```
@Autowired
@Qualifier("MicroTxLRA")
RestTemplate restTemplate;
```

OpenFeign

Configure an interceptor, provided by the MicroTx library, in one of the following ways. This is required if an outgoing request is a part of a transaction or if you want to intercept an outgoing request.

- Configure an interceptor in the configuration property of the `@FeignClient` annotation as shown in the following example.

```
@FeignClient(value = "<feign1.name>", url = "<feign1.url>",
configuration = MicroTxLRAFeignClientInterceptorConfig.class)
public interface <interface_name> {
    //...
}
```

Where, you replace `<feign1.name>` and `<feign1.url>` with values specific to your environment. Note down the name of the interface that you specify as you will use it later in your application code.

- Configure interceptors by specifying property values as shown in the following example. Provide these values in the same file that contains values for the MicroTx client library properties.

```
spring.cloud.openfeign.client.config.<participant1_service_name>.request-
interceptors[0]=
com.oracle.microtx.springboot.lra.clientinterceptor.MicroTxLRAFeignClien
tInterceptorConfig
```

Where, you replace `<feign1.name>` and `<feign1.url>` with values specific to your environment. Note down the name of the interface as you will use it later in your application code.

5. Optional. Perform the following steps only if you want to execute the methods in the Spring Boot participant applications asynchronously. Instead of waiting for a called participant's method to be executed completely, and then calling another method, you can use the `Async` annotation to run each method as an isolated thread.

- a. Import the `com.oracle.microtx.springboot.lra.annotation.*` package.

```
import com.oracle.microtx.springboot.lra.annotation.*
```

- b. Create an executor bean, with a unique name, to override the `SimpleAsyncTaskExecutor` class available in the Spring framework. While creating the executor bean, you must set the task decorator named `MicroTxTaskDecorator`, which is part of the MicroTx Distributed Transactions library. The following example shows how you can create a bean named `taskExecutorForTripBooking`, and then set `MicroTxTaskDecorator`, the MicroTx Distributed Transactions task decorator. Perform this task for each method that you want to run asynchronously.

Example Code

```
@Bean(name = "taskExecutorForTripBooking")
public Executor asyncExecutor()
{
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
```

```

        ... // application-specific code
        executor.setTaskDecorator(new MicroTxTaskDecorator());
        ... // application-specific code
        return executor;
    }

```

Note down the name of the bean as you will need to provide this name as an attribute in the `@Async` annotation.

If your application code already contains a task decorator, you can add the logic from the `MicroTxTaskDecorator` class into your existing task decorator, instead of creating a new one.

- c. Annotate the methods that you want to execute in a separate thread with `@Async`. For value, specify the name of the executor bean that you have created in the previous step as shown in the following example.

Example Code

```

@Async(value = "taskExecutorForTripBooking")
@Override
public CompletableFuture bookHotel(String name, String id)
{
    //Code to implement the application's business logic
}

```

Source code of a sample Spring REST-based transaction initiator application which uses the MicroTx Distributed Transactions library is available in the [trip-manager-springboot](#) folder in the `microtx-samples` GitHub repository. Source code of a sample Spring REST-based transaction participant application which uses the MicroTx Distributed Transactions library is available in the [hotel-springboot](#) folder in the `microtx-samples` GitHub repository. You can use these files as a reference while integrating the MicroTx libraries with your application.

4.3.4 Develop Micronaut Apps with Saga

You can integrate the MicroTx Distributed Transactions library files with Micronaut applications that use the Saga transaction protocol.

For reference information about the MicroTx Distributed Transactions library for Micronaut applications, see [Oracle® Transaction Manager for Microservices Micronaut API Reference for Saga](#).

To integrate the MicroTx Distributed Transactions library files with your Micronaut applications:

1. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the Micronaut application's `pom.xml` file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.

```

<dependency>
    <groupId>com.oracle.microtx.lra</groupId>
    <artifactId>microtx-lra-micronaut</artifactId>
    <version>26.1</version>
</dependency>

```

2. Specify values for the MicroTx Distributed Transactions library properties in the `application.properties` file of your application. This enables the custom library to establish communication with the MicroTx Distributed Transactions Saga coordinator, participate in Saga transactions, and propagate the relevant headers for the coordinated

transactions. See [Configure MicroTx Distributed Transactions Library Properties for Java Apps](#).

3. Import the `com.oracle.microtx.lra.annotation.*` package.

```
import com.oracle.microtx.lra.annotation.*
```

Source code of a sample Micronaut transaction initiator application which uses the MicroTx Distributed Transactions library is available in the [trip-manager-micronaut](#) folder in the `microtx-samples` GitHub repository. Source code of a sample Micronaut transaction participant application which uses the MicroTx Distributed Transactions library is available in the [flight-micronaut](#) folder in the `microtx-samples` GitHub repository. You can use these files as a reference while integrating the MicroTx libraries with your application.

4.3.5 Develop Java Apps that Use Saga and Lock-Free Reservation

MicroTx Distributed Transactions Saga libraries for Spring REST and Micronaut applications support lock-free reservation.

For reference information about the MicroTx Distributed Transactions library for Spring REST-based applications, see [Oracle® Transaction Manager for Microservices Spring Boot API Reference for Saga](#).

When microservices use only Oracle Database 23ai as resource manager, you can use Sagas provided by Oracle Database to manage the transaction. If one microservice uses Oracle Database 23ai as resource manager and other microservices use other databases, then you can integrate the MicroTx Distributed Transactions Saga libraries with Spring REST and Micronaut applications to leverage the lock-free reservation feature in Oracle Database 23ai.

The Lock-Free Reservation feature was introduced in Oracle Database 23ai. See *Using Lock-Free Reservation in Database Development Guide*.

1. Set the property values for the MicroTx Distributed Transactions client library. To enable lock-free reservation, you must set to true the `spring.microtx.lra.lock-free-reservation-active` property for Spring-REST based apps or the `microtx.lra.lock-free-reservation-active` property for Micronaut apps. For details about the other library properties, see [Configure MicroTx Distributed Transactions Library Properties for Java Apps](#).

2. Import the `com.oracle.microtx.lra.lockfree` package.

```
import com.oracle.microtx.lra.lockfree;
```

3. Leverage the MicroTx Distributed Transactions Lock-free API abstraction by injecting an instance of the `MicroTxLockFreeReservation` class in the application code before your application logic begins or joins a transaction. The `MicroTxLockFreeReservation` class implements the `LockFreeReservation` interface.

- For Spring-REST based apps, add the following lines of code.

```
@Autowired  
private MicroTxLockFreeReservation microTxLockFreeReservation;
```

- For Micronaut apps, add the following lines of code.

```
@Inject
private MicroTxLockFreeReservation microTxLockFreeReservation;
```

4. Modify the code of the application that *initiates* a transaction to demarcate the transaction boundaries and to begin a transaction.

```
microTxLockFreeReservation.begin(connection);
... // Business logic to begin a transaction.
```

Where, `connection` is the name of the database connection that your application uses to read and write application data to Oracle Database.

5. Modify the code of the application that *participates* in the transaction to demarcate the transaction boundaries and to join a transaction.

```
microTxLockFreeReservation.join(connection);
... // Business logic to join an existing transaction.
```

Where, `connection` is the name of the database connection that your application uses to read and write application data to Oracle Database.

6. Based on your business logic, complete or compensate the transaction.

- To complete a transaction:

```
microTxLockFreeReservation.complete(connection);
```

- To compensate a transaction:

```
microTxLockFreeReservation.compensate(connection);
```

Where, `connection` is the name of the database connection that your application uses to read and write application data to Oracle Database.

The following sample code demonstrates how you can demarcate the transaction boundaries using `join()`, `complete()`, and `compensate()` for Spring REST-based application that participates in a transaction. The code lines in bold indicate the lines of code that are added to the existing application code to integrate the MicroTx Distributed Transactions library. The sample code has been truncated with ellipses (...) for readability. You can view the entire sample code in the `lra/lockfree` folder in the [microtx-samples](#) GitHub repository.

```
import com.oracle.microtx.lra.lockfree;
...
// code specific to application's business logic
...
//Create an instance of the MicroTxLockFreeReservation class
//which implements the LockFreeReservation interface
@Autowired
private MicroTxLockFreeReservation microTxLockFreeReservation;
...
@RequestMapping(value="/APPLICATION-URL", method = RequestMethod.POST)
@LRA(value=LRA.Type.MANDATORY, end = false)
....
```

```
// In the transaction participant application code,
// provide business logic for the join method so that the
// participant app can join an existing transaction, that MicroTx coordinates.
microTxLockFreeReservation.join(connection);

@RequestMapping(value="/complete", method = RequestMethod.PUT)
@Complete
// application specific business logic to complete the transaction
...

//In transaction initiator and transaction participant application code,
// provide business logic for the complete and compensate methods so that the
// that MicroTx coordinator can call these methods.
microTxLockFreeReservation.complete(connection);
...

@RequestMapping(value="/compensate", method = RequestMethod.PUT)
@Compensate
// application specific business logic to compensate the transaction
...
//
microTxLockFreeReservation.compensate(connection);
...
```

In a similar way, you can demarcate the transaction boundaries for a transaction initiator application. The only change is that you call the `begin()` method to initiate the transaction and you won't call the `join()` method, which is used only by the participant application.

Source code of a sample Spring REST-based transaction initiator application which uses the MicroTx Distributed Transactions library is available in the [trip-manager-springboot](#) folder in the `microtx-samples` GitHub repository. Source code of a sample Spring REST-based transaction participant application which uses the MicroTx Distributed Transactions library is available in the [hotel-springboot](#) folder in the `microtx-samples` GitHub repository. You can use these files as a reference while integrating the MicroTx libraries with your application to leverage the lock-free reservation feature in Oracle Database 23ai.

4.4 Develop Node.js Apps with Saga

The MicroTx Distributed Transactions library for Node.js provides the functionality to initiate a new Saga transaction or to participate in an existing Saga transaction. You must integrate this library into your Node.js application code.

For information about the MicroTx Distributed Transactions library, see [Oracle® Transaction Manager for Microservices TypeScript API Reference](#).

- [Configure Library Properties for Node.js Apps](#)
Provide configuration information for the MicroTx Distributed Transactions library properties. You must perform this step for all the Node.js applications which participate or initiate the transaction.
- [Configure Node.js Apps as Transaction Initiator or Participant](#)

4.4.1 Configure Library Properties for Node.js Apps

Provide configuration information for the MicroTx Distributed Transactions library properties. You must perform this step for all the Node.js applications which participate or initiate the transaction.

Open the `tmm.properties` file in any code editor, and then enter values for the following parameters to configure the MicroTx Distributed Transactions library.

- `oracle.tmm.TcsUrl`: Enter the URL to access the MicroTx Distributed Transactions Saga coordinator. This permits the MicroTx Distributed Transactions library to communicate and coordinate transactions with the coordinator. To get the URL, append `/lra-coordinator` to the URL to access MicroTx Distributed Transactions. For example, `https://192.0.2.1:443/api/v1/lra-coordinator`. For information to identify the URL to access MicroTx Distributed Transactions, see [Access MicroTx Distributed Transactions](#). You must enter this value for the transaction initiator application. You don't have to specify this value for the transaction participant applications.
- `oracle.tmm.CallbackUrl`: Enter the URL of your participant service associated with the MicroTx Distributed Transactions client library. MicroTx Distributed Transactions uses the URL that you provide to connect to the participant service. Provide this value in the following format:

```
https://externalHostnameOfApp:externalPortOfApp/
```

Where,

- `externalHostnameOfApp`: The external host name of your initiator or participant service. For example, `bookTicket-app`.
- `externalPortOfApp`: The port number over which you can access your initiator or participant service remotely. For example, `8081`.

You must specify this value for the transaction participant applications.

- `oracle.tmm.PropagateTraceHeaders`: Set this to `true` when you want to trace the transaction from end-to-end. This propagates the Saga context headers through non-LRA endpoints for all incoming and outgoing requests. This ensures proper coordination with other Saga participants.

For example,

```
oracle.tmm.TcsUrl = https://192.0.2.1:443/api/v1/lra-coordinator
oracle.tmm.CallbackUrl = https://bookTicket-app:8081
oracle.tmm.PropagateTraceHeaders = true
```

You can use the HTTP protocol if your application and MicroTx Distributed Transactions are in the same Kubernetes cluster, otherwise use the HTTPS protocol.

You can also provide these configuration values as environment variables. Note that if you specify values in both the properties file as well as the environment variables, then the values set in the environment variables override the values in the properties file.

The following example provides sample values to configure the environment variables.

```
export ORACLE_TMM_TCS_URL = http://tmm-app:9000/api/v1
export ORACLE_TMM_CALLBACK_URL = http://bookTicket-app:8081
export ORACLE_TMM_PROPAGATE_TRACE_HEADERS = true
```

Note that the environment variables names are case-sensitive.

4.4.2 Configure Node.js Apps as Transaction Initiator or Participant

1. Add the MicroTx Distributed Transactions library for Node.js as a dependency in the `package.json` file.

```
"dependencies": {
  "@oracle/microtx": "<version>"
}
```

Where, `<version>` is the latest version of the MicroTx Distributed Transactions library for Node.js.

2. Configure the property values for the MicroTx Distributed Transactions library. See [Configure Library Properties for Node.js Apps](#).
3. To enable logging for Node.js applications, you must set additional properties. See [Enable Logs for MicroTx Distributed Transactions Node.js Library](#).
4. Configure the MicroTx Distributed Transactions library properties for the microservice by passing the `tmm.properties` file in which you have defined the values.

```
TrmConfig.init('./tmm.properties');
```

5. Import the MicroTx Distributed Transactions and Express libraries.

```
import { Request, Response, Router } from 'express';
import { getLRAId, LRA, LRAConfig, LRAType, ParticipantStatus, cancelLRA,
  LRA_HTTP_CONTEXT_HEADER, LRA_HTTP_ENDED_CONTEXT_HEADER } from "@oracle/
  microtx/lra/lra";
import { getHeaderValue } from '@oracle/microtx/util/trmutils';
```

6. Create a router object to handle requests in your program.

Use the following code to create a router object named `flightSvcRouter`.

```
const flightSvcRouter = Router();
```

7. Enter the URL of the MicroTx Distributed Transactions Saga coordinator. To get this attribute value, append `/lra-coordinator` to the URL that you use to access MicroTx Distributed Transactions. For example, if `https://tmm-app:9000/api/v1` is the MicroTx Distributed Transactions URL, then `lraCoordinatorUrl` is `https://tmm-app:9000/api/v1/lra-coordinator`.

```
const lraCoordinateUrl = process.env.ORACLE_TMM_TCS_URL
```

8. Add the following code to initialize the LRAConfig object for the REST endpoints of the transaction initiator and transaction participant services. The services may expose many REST API endpoints, but you have to initialize LRAConfig object only for the REST API endpoints which need to participate in the Saga transaction.

```
const lra: LRA = new LRA("/flight", LRAType.REQUIRES_NEW);
lra.end = false;
lra.timeLimitInMilliseconds = 100000;
new LRAConfig(lraCoordinateUrl, flightSvcRouter, "/flightService/api",
lra, "/complete", "/compensate", "/status", "/after", "", "", "");
```

Where,

- */flight* is the REST API endpoint which the transaction initiator application exposes to participate in the Saga transaction.
- `LRAType.REQUIRES_NEW` determines if the service participates in an existing Saga transaction or creates a new one. When you set `LRAType` as `REQUIRES_NEW`, a new transaction is created. When you set `LRAType` as `MANDATORY`, the service participates in an existing transaction. For details about the `LRAType` values, see [Transaction Manager for Microservices TypeScript API Reference](#).
- *flightSvcRouter* is the router object that you have created previously.
- */flightService/api* is the mount point of the *flightSvcRouter* router. This is the value for the `applRouterMountPath` field of the `LRAConfig` object.
- `timeLimitInMilliseconds` is the time period, in milliseconds, within which the transaction must be completed or compensated. If the transaction is not completed within the specified time period, MicroTx Distributed Transactions compensates the transaction. Decide the time limit based on your business requirement.
- `"/complete", "/compensate", "/status", "/after"` are the REST API endpoints for which you want to define your application's business logic.

During the Saga transaction, MicroTx Distributed Transactions adds a link header for all the outgoing requests from the REST API endpoints that you have specified.

9. Define the business logic for all the REST API endpoints that you have mentioned while creating the `LRAConfig` object.

```
flightSvcRouter.put('/complete', async (req, resp) => {
//application business logic
});

flightSvcRouter.put('/compensate', async (req, resp) => {
//application business logic
});

flightSvcRouter.put('/status', async (req, resp) => {
//application business logic
});

flightSvcRouter.put('/after', (req, resp) => {
//application business logic
});
```

Where, *flightSvcRouter* is the router object that you have created previously. Although the sample code mentions only put, you can use any HTTP verb based on your business logic.

10. Save your changes.

Source code of a sample Node.js transaction participant application which uses the MicroTx Distributed Transactions library is available in the [flight](#) folder in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

5

Develop Applications with TCC

The MicroTx Distributed Transactions library provides applications the functionality to initiate a new TCC transaction or to participate in an existing TCC transaction.

Topics

- [Guidelines](#)
Here are a few guidelines to develop custom applications that use the TCC transaction protocol.
- [Workflow to Develop Applications with TCC](#)
Use the following workflow as a guide to develop your applications to use MicroTx Distributed Transactions to manage TCC transactions.
- [Configure Library Properties](#)
Provide configuration information for the MicroTx Distributed Transactions client library properties. You must perform this step for all participant and initiator applications.
- [About Transaction Timeout](#)
Specify the time period for which a request remains active. This value is specific to each microservice that participates in a TCC transaction. If a transaction is not confirmed or canceled by a microservice within the specified time period, the transaction is canceled.
- [Develop Spring Boot Apps with TCC](#)
Use the MicroTx Distributed Transactions library with your Spring Boot applications.
- [Develop JAX-RS Apps with TCC](#)
The MicroTx Distributed Transactions library intercepts the incoming HTTP calls using JAX-RS filters, and then initiates a new TCC transaction or joins an existing transaction.
- [Develop Node.js Apps with TCC](#)
Use the MicroTx Distributed Transactions library with your Node.js applications.
- [Develop Python Apps with TCC](#)
MicroTx Distributed Transactions client libraries for Python applications provides the functionality to initiate a new TCC transaction or to participate in an existing TCC transaction.

5.1 Guidelines

Here are a few guidelines to develop custom applications that use the TCC transaction protocol.

In the TCC protocol, a transaction initiator services asks other participant microservices to reserve resources. When the initiator and all participants have acquired the required reservations, the initiator then sends a request to MicroTx Distributed Transactions to confirm all the reservations.

Based on its business logic, if the initiator service decides that it does not want or cannot use the reservations made, it requests the MicroTx Distributed Transactions to cancel all the reservations. What constitutes a reservation is completely up to the application.

The TCC transaction protocol relies on the basic HTTP verbs: POST, PUT, and DELETE. Ensure that your application conforms to the following guidelines:

- The transaction initiator service must use the POST HTTP method to create a new reservation. As a response to this request, the transaction participant services must return a URI representing the reservation. The MicroTx Distributed Transactions client libraries places the URI in MicroTx Distributed Transactions specific headers to ensure that the URI is propagated up the call stack.
- This protocol relies upon the participant services to ensure that all participant services either confirm their reservations or cancel their reservations. The URIs must respond to the PUT HTTP method to confirm a reservation, and to the DELETE HTTP method to cancel a reservation.

5.2 Workflow to Develop Applications with TCC

Use the following workflow as a guide to develop your applications to use MicroTx Distributed Transactions to manage TCC transactions.

Task	Description	More Information
Install MicroTx Distributed Transactions	Install MicroTx Distributed Transactions and ensure that you can access it.	Workflow to Install MicroTx
Provide configuration information for the MicroTx Distributed Transactions library properties.	Perform this step for all the transaction participant and transaction initiator applications so that your applications can access the library.	Configure Library Properties
Integrate the MicroTx Distributed Transactions library with your application code.	Select a suitable procedure to integrate the library based on the following factors: • the development framework for your application • whether an application initiates the transaction or participates in the transaction	The library is available for Java, Node.js, and Python apps. Perform one of the following tasks: • Develop JAX-RS Apps with TCC • Develop Node.js Apps with TCC • Develop Python Apps with TCC
Deploy your application	Develop, test, and deploy your microservices independently. After integrating the library files with your application, deploy the application.	Deploy Your Application

5.3 Configure Library Properties

Provide configuration information for the MicroTx Distributed Transactions client library properties. You must perform this step for all participant and initiator applications.

For JAX-RS and Node.js applications, open the `tmm.properties` file in any code editor, and then enter values for the following parameters to configure the MicroTx library. For Spring REST applications, use the `application.yaml` file.

The name of the properties differ for JAX-RS and Spring REST applications, but the functionality remains the same.

- `spring.microtx.coordinator-url` for Spring REST apps and `oracle.tmm.TcsUrl` for other apps: Enter the URL to access the MicroTx Distributed Transactions application. See [Access MicroTx Distributed Transactions](#). You must enter this value for the transaction initiator application. You don't have to specify this value for the transaction participant applications.
- `spring.microtx.propagation-active` for Spring REST apps and `oracle.tmm.PropagateTraceHeaders` for other apps: Set this to true when you want to

trace the transaction from end-to-end. This propagates the trace headers for all incoming and outgoing requests. For Helidon-based microservices, set this property to false to avoid propagating the trace headers twice as Helidon framework propagates trace headers by default. You can set this property to true if propagation of trace headers is disabled in Helidon configuration and you want to enable distributed tracing with MicroTx Distributed Transactions. For other microservices, set this property to true.

The following example provides sample values for the MicroTx Distributed Transactions client library properties for a Spring REST application.

```
spring:
  microtx:
    coordinator-url: http://tmm-app:9000/api/v1
    propagation-active: true
```

The following example provides sample values for the MicroTx Distributed Transactions client library properties for others apps, such as JAX-RS, Python, or Node.js applications.

```
oracle.tmm.TcsUrl = http://tmm-app:9000/api/v1
oracle.tmm.PropagateTraceHeaders = true
```

You can use the HTTP protocol if your application and MicroTx Distributed Transactions are in the same Kubernetes cluster, otherwise use the HTTPS protocol.

You can also provide these configuration values as environment variables. Note that if you specify values in both the properties file as well as the environment variables, then the values set in the environment variables override the values in the properties file.

The following example provides sample values to configure the environment variables for Spring REST apps.

```
export SPRING_MICROTX_COORDINATOR_URL = http://tmm-app:9000/api/v1
export SPRING_MICROTX_PROPAGATION_ACTIVE = true
```

The following example provides sample values to configure the environment variables for others apps, such as JAX-RS, Python, or Node.js applications.

```
export ORACLE_TMM_TCS_URL = http://tmm-app:9000/api/v1
export ORACLE_TMM_PROPAGATE_TRACE_HEADERS = true
```

Note that the environment variables names are case-sensitive.

5.4 About Transaction Timeout

Specify the time period for which a request remains active. This value is specific to each microservice that participates in a TCC transaction. If a transaction is not confirmed or canceled by a microservice within the specified time period, the transaction is canceled.

In a TCC transaction, the transaction initiator service collects the status of reservations of all the participant services and decides whether the transaction should be confirmed or canceled, MicroTx Distributed Transactions ensures that all participant services either confirm or cancel the reservation. When MicroTx Distributed Transactions sends a request to confirm the transaction, some participant services may confirm the transaction while the transaction may time out for other participant services. It is the responsibility of the application developer to

provide the required code to cancel the reservations and release the resources in case the transaction times out. MicroTx Distributed Transactions sends a request to all participant services to either confirm or cancel the reservation based on the decision taken by the transaction initiator's business logic.

5.5 Develop Spring Boot Apps with TCC

Use the MicroTx Distributed Transactions library with your Spring Boot applications.

For information about the MicroTx Distributed Transactions library, see [Oracle® Transaction Manager for Microservices Spring Boot API Reference for XA and TCC](#).

Topics

- [Configure Spring REST App as Transaction Initiator](#)
- [Configure Spring Boot App as Transaction Participant](#)

5.5.1 Configure Spring REST App as Transaction Initiator

1. Specify property values for the MicroTx Distributed Transactions library. See [Configure Library Properties](#).
2. Include the following MicroTx Distributed Transactions library file as a maven dependency in the JDBC application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

```
<dependency>
  <groupId>com.oracle.microtx</groupId>
  <artifactId>microtx-spring-boot-starter</artifactId>
  <version>26.1</version>
</dependency>
```

3. Import the following packages.

```
import com.oracle.microtx.tcc.MicroTxTccClient;
import com.oracle.microtx.tcc.annotation.TCC;
import oracle.tmm.tcc.TccParticipantStatus;
import oracle.tmm.tcc.exception.TccException;
import oracle.tmm.tcc.exception.TccHeuristicException;
import oracle.tmm.tcc.exception.TccUnknownTransactionException;
import oracle.tmm.tcc.vo.CancelResponse;
import oracle.tmm.tcc.vo.ConfirmResponse;
import oracle.tmm.tcc.vo.TccParticipant;
```

4. Add @TCC annotation before the initiator application resource class. This initiates a new TCC transaction and adds a header for all the outgoing REST API requests from the transaction initiator.

Use the following code to initiate a new TCC transaction when a call is made to the transaction initiator service. In the following example, the class

myTransactionInitiatorApp contains the code that initiates the service. Replace the name of the class based on your environment.

```
@TCC(timeLimit = 120, timeUnit = ChronoUnit.SECONDS) //Add @TCC
annotation before the initiator application resource class to start a TCC
transaction
public class myTransactionInitiatorApp {
    // Service code that is specific to the transaction initiator service.
}
```

You can specify the following optional parameters with the @TCC annotation.

- **timeLimit**: Specify the time period, as a whole number, for which you want the transaction initiator service to reserve the resources. It is the responsibility of the application developer to provide the required code to release the resources and cancel their part of the TCC transaction after the time limit expires. Decide the time limit based on your business requirement.
 - **timeUnit**: Specify the unit in which you have mentioned the time limit, such as `ChronoUnit.SECONDS` and `ChronoUnit.MINUTES`. Permissible values are all the enum values from the `java.time.temporal.ChronoUnit` class. See <https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html>.
5. Create an object, of a class in the MicroTx Distributed Transactions library, to call the endpoints of other TCC transaction participant services. The transaction initiator service begins the transaction. To complete the transaction, the initiator service may have to make calls to one or more participant services. While calling the participant services, use the object that you create.

-
- [RestTemplate](#)
 - [OpenFeign](#)

RestTemplate

Autowire the Spring Boot RestTemplate, provided by the MicroTx Distributed Transactions library.

```
@Autowired
@Qualifier("MicroTxTccRestTemplate")
RestTemplate restTemplate;
```

OpenFeign

Configure an interceptor, provided by the MicroTx Distributed Transactions library, in one of the following ways. This is required if an outgoing request is a part of a transaction or if you want to intercept an outgoing request.

- Configure an interceptor in the configuration property of the @FeignClient annotation as shown in the following example.

```
@FeignClient(value = "<feign1.name>", url = "<feign1.url>",
configuration = MicroTxTccFeignInterceptorConfig.class)
public interface <interface_name> {
```

```
// ...
}
```

Where, you replace `<feign1.name>` and `<feign1.url>` with values specific to your environment. Note down the name of the interface as you will use it later in your application code.

- Configure interceptors by specifying property values as shown in the following example. Provide these values in the same file that contains values for the MicroTx Distributed Transactions client library properties.

```
spring.cloud.openfeign.client.config.<feign1_name>.request-
interceptors[0]=
com.oracle.microtx.springboot.tcc.clientinterceptor.MicroTxTccFeignInter-
ceptorConfig
```

Where, `<feign1_name>` is the data for the value property of the `@FeignClient` annotation.

6. In the transaction initiator application code, inject a `MicroTxTccClient` object, and then use the injected object to confirm or cancel the transaction. The following code example describes the changes that you need to make to the initiator application code.

- [RestTemplate](#)
- [OpenFeign](#)

RestTemplate

```
@TCC
public class myTransactionParticipantApp {
    // Service code that is specific to the transaction participant
    service.

    @Autowired
    @Qualifier("MicroTxTccRestTemplate")
    RestTemplate restTemplate;

    @Autowired
    MicroTxTccClient tccClientService;
    ...
}
```

OpenFeign

```
@TCC
public class myTransactionParticipantApp {
    // Service code that is specific to the transaction participant
```

```

service.

    @Autowired
    MicroTxTccClient tccClientService;
    ...
}

```

7. Save the changes.

Source code of a sample Spring REST transaction initiator application which uses the MicroTx Distributed Transactions library is available in the [travel-agent-springboot](#) folder which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

5.5.2 Configure Spring Boot App as Transaction Participant

1. Specify property values for the MicroTx Distributed Transactions library. See [Configure Library Properties](#).
2. Include the following MicroTx Distributed Transactions library file as a maven dependency in the JDBC application's `pom.xml` file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

```

<dependency>
    <groupId>com.oracle.microtx</groupId>
    <artifactId>microtx-spring-boot-starter</artifactId>
    <version>26.1</version>
</dependency>

```

3. Import the following packages.

```

import com.oracle.microtx.tcc.MicroTxTccClient;
import com.oracle.microtx.tcc.annotation.TCC;

```

4. Inject TCC annotation in the transaction participant application code.

To enable participant services join an existing TCC transaction, add `@TCC` annotation before the resource class of the transaction participant service. Insert the following code in the transaction participant code. In the following example, the `myTransactionParticipantApp` class contains code for the transaction participant service. Replace the name of the class based on your environment.

```

@Path("/")
@TCC(timeLimit = 120, timeUnit = ChronoUnit.SECONDS)
//Add @TCC annotation so that the transaction participant service joins an
existing TCC transaction
//The transaction initiator service passes the TCC context in the request
header.
public class myTransactionParticipantApp {
    // Service code that is specific to the transaction participant
    service.
}

```

5. Autowire the Spring Boot RestTemplate, provided by the MicroTx library. Use this object to call the endpoints of other TCC transaction participant services.

```
@Autowired
@Qualifier("MicroTxTccRestTemplate")
RestTemplate restTemplate;
```

6. In the transaction participant application code, inject a MicroTxTccClient object and then call the addTccParticipant(String uri) method to register a participant service with the TCC transaction. The participant service exposes a URI which MicroTx uses to confirm or cancel the transaction. MicroTx calls the PUT method to confirm the transaction and the DELETE method to cancel the transaction and release the reserved resource. Ensure that these methods are present and the confirm and cancel logic is implemented. To confirm or cancel the transaction, MicroTx sends a call to the exposed URI of all the participant services.

The following code example describes the changes that you need to make to the participant application code.

```
@TCC
public class myTransactionParticipantApp {
    // Service code that is specific to the transaction participant
    service.

    @Autowired
    @Qualifier("MicroTxTccRestTemplate")
    RestTemplate restTemplate;

    @Autowired
    MicroTxTccClient tccClientService;

    @POST
    //The REST endpoint of the transaction participant service.
    @Path("bookings")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response create() throws TccUnknownTransactionException {
        // Business logic to create a booking.
        String bookingUri;
        // Register participant service with the TCC transaction
        tccClientService.addTccParticipant(bookingUri.toString());
    }

    @PUT
    @Path("bookings/{bookingId}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response confirm() throws TccUnknownTransactionException {
        //Application-specific code to confirm the booking.
    }

    @DELETE
    @Path("bookings/{bookingId}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response cancel() throws TccUnknownTransactionException {
        //Application-specific code to cancel the booking.
    }
}
```

```
    }  
}
```

Where,

- *myTransactionParticipantApp* is a class that contains code for the transaction participant service. This class already contains user-defined methods that the participant service uses to confirm or cancel a transaction.
- *bookings* is the REST endpoint of the transaction participant service. The transaction initiator service calls this endpoint to perform a task, such as creating a hotel booking.
- *bookingUri* contains the resource URI that the participant service exposes and which MicroTx Distributed Transactions uses to confirm or cancel the transaction.
- *bookingId* is the unique ID of the booking that you want to confirm or cancel.

7. Save the changes.

Ensure that you make these changes in the code of all transaction participant applications.

Source code of a sample Spring REST transaction participant application which uses the MicroTx Distributed Transactions library is available in the [flight-booking-springboot](#) folder which is located in the *microtx-samples* GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

5.6 Develop JAX-RS Apps with TCC

The MicroTx Distributed Transactions library intercepts the incoming HTTP calls using JAX-RS filters, and then initiates a new TCC transaction or joins an existing transaction.

Use the following annotation to add TCC functionality to your application code and enlist the participant services.

- `@TCC(timeLimit = 120, timeUnit = ChronoUnit.SECONDS)`
Use this to annotate the application-specific REST resource that you want MicroTx Distributed Transactions to call to initiate a new TCC transaction or join an existing transaction.

When you add an annotation to a class, the JAX-RS filters look for the annotation to identify the class that participates in the TCC transaction. If the request header does not contain a value for *link*, then the MicroTx Distributed Transactions library creates a value for *link* in the request header and a unique transaction ID. You can use the unique transaction ID to identify, trace, or debug the transaction.

If the request header contains value for *link*, then the application participates in the existing TCC transaction. All the applications that participate in the transaction share a unique TCC transaction ID. Here's an example value for *link* in the request header:

```
link=[<http://tmm-app:9000/api/v1/tcc-transaction/7ff...>; rel="https://otmm.oracle.com/tcc-transaction/internal",<http://tmm-app:9000/api/v1/tcc-transaction/7ff...>; rel="https://otmm.oracle.com/tcc-transaction/external"]
```

Where, *7ff...* is the unique transaction ID. Example values have been truncated with ... to improve readability. When you view the header in your environment, you'll see the entire value.

For information about the MicroTx Distributed Transactions library, see [Oracle® Transaction Manager for Microservices Java \(Jakarta\) API Reference](#).

Topics

- [Configure JAX-RS App as Transaction Initiator](#)
- [Configure JAX-RS App as Transaction Participant](#)

5.6.1 Configure JAX-RS App as Transaction Initiator

Before you begin, ensure that you have configured the property values for the MicroTx Distributed Transactions library.

1. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's pom.xml file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.

- In Jakarta EE8 environments, such as Helidon 2.x, use the TmmLib file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib</artifactId>
  <version>26.1</version>
</dependency>
```

- In Jakarta EE9 environments, such as Helidon 3.x applications, use the TmmLib-jakarta file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib-jakarta</artifactId>
  <version>26.1</version>
</dependency>
```

2. Import the following packages.

```
import oracle.tmm.tcc.TccClientService;
import oracle.tmm.tcc.TccParticipantStatus;
import oracle.tmm.tcc.TccParticipantStatus;
import oracle.tmm.tcc.exception.TccException;
import oracle.tmm.tcc.exception.TccHeuristicException;
import oracle.tmm.tcc.exception.TccUnknownTransactionException;
import oracle.tmm.tcc.vo.CancelResponse;
import oracle.tmm.tcc.vo.ConfirmResponse;
import oracle.tmm.tcc.vo.TccParticipant;
```

3. Add @TCC annotation before the initiator application resource class. This initiates a new TCC transaction and adds a header for all the outgoing REST API requests from the transaction initiator.

Use the following code to initiate a new TCC transaction when a call is made to the transaction initiator service. In the following example, the class *myTransactionInitiatorApp* contains the code that initiates the service. Replace the name of the class based on your environment.

```
@TCC(timeLimit = 120, timeUnit = ChronoUnit.SECONDS) //Add @TCC
annotation before the initiator application resource class to start a TCC
```

```
transaction
public class myTransactionInitiatorApp {
    // Service code that is specific to the transaction initiator service.
}
```

You can specify the following optional parameters with the @TCC annotation.

- **timeLimit**: Specify the time period, as a whole number, for which you want the transaction initiator service to reserve the resources. It is the responsibility of the application developer to provide the required code to release the resources and cancel their part of the TCC transaction after the time limit expires. Decide the time limit based on your business requirement.
- **timeUnit**: Specify the unit in which you have mentioned the time limit, such as `ChronoUnit.SECONDS` and `ChronoUnit.MINUTES`. Permissible values are all the enum values from the `java.time.temporal.ChronoUnit` class. See <https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html>.

Source code of a sample JAX-RS transaction initiator application which uses the MicroTx Distributed Transactions library is available in the [BookingResource.java](#) file which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

5.6.2 Configure JAX-RS App as Transaction Participant

Before you begin, ensure that you have configured the property values for the MicroTx Distributed Transactions library.

1. Include the MicroTx Distributed Transactions Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 26.1 release. Provide the correct version, based on the release version that you want to use.

- In Jakarta EE8 environments, such as Helidon 2.x, use the `TmmLib` file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib</artifactId>
  <version>26.1</version>
</dependency>
```

- In Jakarta EE9 environments, such as Helidon 3.x applications, use the `TmmLib-jakarta` file.

```
<dependency>
  <groupId>com.oracle.tmm.jta</groupId>
  <artifactId>TmmLib-jakarta</artifactId>
  <version>26.1</version>
</dependency>
```

2. Import the following packages.

```
import oracle.tmm.tcc.annotation.TCC;
import javax.ws.rs.core.Application;
```

3. Inject TCC annotation in the transaction participant application code.

To enable participant services join an existing TCC transaction, add `@TCC` annotation before the resource class of the transaction participant service.

Insert the following code in the transaction participant code. In the following example, the *myTransactionParticipantApp* class contains code for the transaction participant service. Replace the name of the class based on your environment.

```
@Path("/")
@TCC(timeLimit = 120, timeUnit = ChronoUnit.SECONDS)
//Add @TCC annotation so that the transaction participant service joins an
existing TCC transaction
//The transaction initiator service passes the TCC context in the request
header.
public class myTransactionParticipantApp extends Application {
    // Service code that is specific to the transaction participant
    service.
}
```

4. In the transaction participant application code, inject a `tccClientService` object and then call the `addTccParticipant(String uri)` method on this object to register a participant service with the TCC transaction. The participant service exposes a URI which MicroTx Distributed Transactions uses to confirm or cancel the transaction. MicroTx Distributed Transactions calls the PUT method to confirm the transaction and the DELETE method to cancel the transaction and release the reserved resource. Ensure that these methods are present and the confirm and cancel logic is implemented. To confirm or cancel the transaction, MicroTx Distributed Transactions sends a call to the exposed URI of all the participant services.

The following code example describes the changes that you need to make to the participant application code.

```
public class myTransactionParticipantApp extends Application {
    // Service code that is specific to the transaction participant
    service.

    @Inject
    TccClientService tccClientService;

    @POST
    //The REST endpoint of the transaction participant service.
    @Path("bookings")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response create() throws TccUnknownTransactionException
    // Business logic to create a booking.
    String bookingUri;
    // Register participant service with the TCC transaction
    tccClientService.addTccParticipant(bookingUri.toString());
}

    @PUT
    @Path("bookings/{bookingId}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response confirm() throws TccUnknownTransactionException {
    //Application-specific code to confirm the booking.
    }
```

```
@DELETE
@Path("bookings/{bookingId}")
@Consumes(MediaType.APPLICATION_JSON)
public Response cancel() throws TccUnknownTransactionException {
    //Application-specific code to cancel the booking.
}
}
```

Where,

- *myTransactionParticipantApp* is a class that contains code for the transaction participant service. This class already contains user-defined methods that the participant service uses to confirm or cancel a transaction.
- *bookings* is the REST endpoint of the transaction participant service. The transaction initiator service calls this endpoint to perform a task, such as creating a hotel booking.
- *bookingUri* contains the resource URI that the participant service exposes and which MicroTx Distributed Transactions uses to confirm or cancel the transaction.
- *bookingId* is the unique ID of the booking that you want to confirm or cancel.

5. Save the changes.

Ensure that you make these changes in the code of all transaction participant services. Source code of a sample JAX-RS transaction participant application which uses the MicroTx Distributed Transactions library is available in the [FlightBookingAppResource.java](#) file which is located in the *microtx-samples* GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

5.7 Develop Node.js Apps with TCC

Use the MicroTx Distributed Transactions library with your Node.js applications.

For information about the MicroTx Distributed Transactions library, see [Oracle® Transaction Manager for Microservices TypeScript API Reference](#).

Use the following TCC helper methods to confirm or cancel the transaction. Both initiator and participant services can access the helper methods.

Helper Method	Description
<code>ConfirmTCC(req.headers);</code>	Confirms the current TCC transaction.
<code>CancelTCC(req.headers);</code>	Cancels the current TCC transaction.
<code>GetTCCId(req.headers)</code>	Get the current TCC transaction ID.

Topics

- [Configure Node.js App as Transaction Initiator](#)
- [Configure Node.js App as Transaction Participant](#)

5.7.1 Configure Node.js App as Transaction Initiator

1. Add the MicroTx Distributed Transactions library for Node.js as a dependency in the package.json file.

```
"dependencies": {  
  "@oracle/microtx": "<version>"  
}
```

Where, <version> is the latest version of the MicroTx Distributed Transactions library for Node.js.

2. Configure the property values for the MicroTx Distributed Transactions library. See [Configure Library Properties](#).
3. To enable logging for Node.js applications, you must set additional properties. See [Enable Logs for MicroTx Distributed Transactions Node.js Library](#).
4. Configure the MicroTx Distributed Transactions library properties for the microservice by passing the tmm.properties file in which you have defined the values.

```
TrmConfig.init('./tmm.properties');
```

5. Import the MicroTx Distributed Transactions libraries and the express module files.

```
import {HttpMethod, TrmConfig} from "@oracle/microtx/util/trmutils";  
import {TCCConfig} from "@oracle/microtx/tcc/tcc";  
import {NextFunction, request, Request, Response, Router} from 'express';
```

6. Create a router object.

Use the following code to create a router object named svcRouter.

```
const svcRouter = Router();
```

7. Add the following code to initialize the TCCConfig object for the REST endpoints of the transaction initiator service. The transaction initiator may expose many REST API endpoints, but you have to initialize TCCConfig object only for the REST API endpoints which need to participate in the TCC transaction.

In the following code sample, the transaction initiator application exposes the /bookings REST API endpoint.

```
// Initialize TCCConfig object for all the endpoints which need to  
participant in the TCC transaction.  
const tccConfig: TCCConfig = new TCCConfig("/bookings", svcRouter,  
HttpMethod.POST, 30);
```

Where,

- svcRouter is the router object that you have created in the previous step.
- 30 is the time limit in seconds for the transaction initiator application to reserve the resources. Specify the time period as a whole number. It is the responsibility of the application developer to provide the required code to release the resources and cancel their part of the TCC transaction after the time limit expires. Decide the time limit based on your business requirement.

Replace these values with the values specific to your environment.

When this code is executed, TCC transaction is initiated and MicroTx Distributed Transactions adds a header for all the outgoing requests from the REST API endpoint that you have specified.

Source code of a sample Node.js transaction initiator application which uses the MicroTx Distributed Transactions library is available in the [travelagent.ts](https://github.com/oracle/microtx-samples) file which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

5.7.2 Configure Node.js App as Transaction Participant

Before you begin, ensure that you have configured the property values for the MicroTx Distributed Transactions library.

1. Add the MicroTx Distributed Transactions library for Node.js as a dependency in the `package.json` file.

```
"dependencies": {  
  "@oracle/microtx": "<version>"  
}
```

Where, `<version>` is the latest version of the MicroTx Distributed Transactions library for Node.js.

2. Configure the property values for the MicroTx Distributed Transactions library. See [Configure Library Properties](#).
3. To enable logging for Node.js applications, you must set additional properties. See [Enable Logs for MicroTx Distributed Transactions Node.js Library](#).
4. Configure the MicroTx Distributed Transactions library properties for the microservice by passing the `tmm.properties` file in which you have defined the values.

```
TrmConfig.init('./tmm.properties');
```

5. Import the MicroTx Distributed Transactions libraries and the express module files.

```
import {HttpMethod, TrmConfig} from "@oracle/microtx/util/trmutils";  
import {TCCConfig} from "@oracle/microtx/tcc/tcc";  
import {NextFunction, request, Request, Response, Router} from 'express';
```

6. Create a router object.

Use the following code to create a router object named `svcRouter2`.

```
const svcRouter2 = Router();
```

7. Add the following code to initialize the `TCCConfig` object for the confirm and cancel REST API endpoints of the transaction participant service.

In the following code sample, the transaction participant application exposes the `/bookings` REST API endpoint. The `svcRouter2` is the router object that you have created in the previous step. Replace these values with the values specific to your environment.

```
//Initialize TCCConfig object for all the endpoints which need to  
participant in the TCC transaction
```

```
const tccConfig: TCCConfig = new TCCConfig("/bookings", svcRouter2,
HttpMethod.POST, 30);
```

Where,

- `/bookings` is the REST API endpoint that the transaction participant service exposes.
 - `svcRouter2` is the router object that you have created previously.
8. In the following code sample, the transaction participant service exposes the `/bookings/:bookingId` REST API endpoint to confirm or cancel the transaction. Replace these values with the values specific to your environment. Also ensure that these endpoints are present in the transaction participant service and the confirm and cancel logic is implemented in the code. The `dohotelBooking()`, `doConfirmBooking()`, and `doCancelBooking()` methods contain the business logic for creating a resource, confirming the transaction, and canceling the transaction respectively. Ensure that the business logic is implemented in the code of the transaction participant service and the endpoints are present.

You'll also mention the HTTP method that the REST API endpoint uses. MicroTx Distributed Transactions uses the PUT method to confirm the transaction and the DELETE method to cancel the transaction and release the resources that were reserved for the specified resource URI.

```
svcRouter.post('/bookings', asyncHandler(async (req: Request, res:
Response) => {
    dohotelBooking(req, res); //app-specific code to create a resource
})));
```

```
svcRouter.put('/bookings/:bookingId', asyncHandler(async (req: Request,
res: Response) => {
    doConfirmBooking(req, res); //app-specific code to confirm the
transaction
})));
```

```
svcRouter.delete('/bookings/:bookingId', asyncHandler(async (req: Request,
res: Response) => {
    doCancelBooking(req, res); //app-specific code to cancel the
transaction
})));
```

9. Use the `TCCConfig` object that you have created earlier to register participants (reserved resource URI) to an existing TCC transaction by calling the `addTccParticipant` method with the resource URI.

```
const bookingUri;
tccConfig.addTccParticipant(bookingUri);
```

When this code is executed, the participant service joins an existing TCC transaction when the initiator service calls the participant service. Also the MicroTx Distributed Transactions library enlists the participant service with the URIs you provide for the confirm and cancel endpoints.

Source code of a sample Node.js transaction initiator application which uses the MicroTx Distributed Transactions library is available in the [flight.ts](#) file which is located in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

5.8 Develop Python Apps with TCC

MicroTx Distributed Transactions client libraries for Python applications provides the functionality to initiate a new TCC transaction or to participate in an existing TCC transaction.

To use MicroTx Distributed Transactions to manage a TCC transaction, update your Python application code to integrate the functionality provided by the MicroTx Distributed Transactions client libraries. For information about the MicroTx Distributed Transactions library, see [Oracle® Transaction Manager for Microservices Python API Reference](#).

Use the following TCCClient helper methods to confirm or cancel the transaction. Both initiator and participant services can access the helper methods.

Method	Description
<code>ConfirmTCC(<i>incoming_request_headers</i>);</code>	Confirms the current TCC transaction and returns the HTTP response.
<code>CancelTCC(<i>incoming_request_headers</i>);</code>	Cancels the current TCC transaction and returns the HTTP response.
<code>GetTCCId(<i>incoming_request_headers</i>)</code>	Gets details of the current TCC transaction ID.

Where, `incoming_request_headers` is a dictionary of key-value pairs.

Topics

- [Configure Python App as Transaction Initiator](#)
- [Configure Python App as Transaction Participant](#)

5.8.1 Configure Python App as Transaction Initiator

You can select Flask or Django as the framework for your Python application. This section provides instructions to integrate the MicroTx Distributed Transactions library with the application code of your Python application with Flask framework.

1. Open a terminal in the virtual environment that you have created for your Python application, and then run the following command to install the MicroTx Distributed Transactions library file for Python which is available in the `installation_directory/MicroTx-<version>/lib/python` folder.

```
pip3 install tmmPy-<version>.whl
```

2. Configure the property values for the MicroTx Distributed Transactions library. Create a new file and save it as `tmm.properties`. You must provide values for the following properties.

The following example provides sample values for the properties. Provide the values based on your environment.

```
oracle.tmm.TcsUrl = http://tmm-app:9000/api/v1
oracle.tmm.PropagateTraceHeaders = true
server.port = 8080
oracle.tmm.CallbackUrl = http://localhost:{server.port}
```

For details about each property, see [Configure Library Properties](#).

Note down the name and location of this file as you will have to provide this later when you initialize the `tccConfig` object.

3. Import the MicroTx Distributed Transactions libraries and exceptions. You can use `tcclib.exception` to handle exceptions.

```
from tcclib.tcc import TCCClient, Middleware, http_request, TCCConfig
import tcclib.exception as ex
```

4. Create a Flask instance with middleware. Middleware helps to intercept all the incoming requests received by the Flask instance.

The following sample code creates a Flask instance and a middleware object.

```
# Create a Flask instance with the name of the current module.
app = Flask(__name__)
# Middleware helps to intercept all the incoming requests received by the
Flask application.
app.wsgi_app = middleware(app.wsgi_app)
```

5. Add the following code to initialize the `tccConfig` object for the microservice.

Syntax

```
tccConfig = TCCConfig(filePath=<application_properties_file_path>,
timeLimitInSeconds=<integer>)
```

Sample

```
tccConfig = TCCConfig(filePath="./tmm.properties", timeLimitInSeconds=300)
```

Where,

- `./tmm.properties` is the location of the file in which you have previously defined values for the MicroTx Distributed Transactions library properties for the transaction initiator service.
- `300` is the time limit in seconds for the transaction initiator service to reserve the resources. Specify the time period as a whole number. It is the responsibility of the application developer to provide the required code to release the resources and cancel their part of the TCC transaction after the time limit expires. Decide the time limit based on your business requirement.

Replace these values with the values specific to your environment.

6. The TCC transaction protocol relies on the basic HTTP verbs: POST, PUT, and DELETE. You must expose the REST API endpoints for each HTTP method and map these endpoints to a specific function that executes the business logic. Your application code already contains the business logic to make a new reservation and confirm or cancel the reservation. Use the `app.route` decorator to bind a function in your application to a HTTP verb and URL path.

In the following code sample for a transaction initiator service, the service exposes the REST API endpoints for the different HTTP verbs.

```
//Mandatory. The transaction initiator service must use the
//POST HTTP method to create a new reservation.
@app.route('/travel-agent/api/bookings/reserve', methods=['POST'])
def do_trip_reserve():
    //app-specific code to create a booking

//Mandatory. Use the PUT HTTP method to confirm a reservation.
@app.route('/travel-agent/api/confirm/<trip_booking_id>', methods=['PUT'])
def do_trip_confirm(trip_booking_id):
    //app-specific code to confirm the specified booking ID

//Mandatory. Use the DELETE HTTP method to cancel a reservation.
@app.route('/travel-agent/api/cancel/<trip_booking_id>',
methods=['DELETE'])
def do_trip_cancel(trip_booking_id):
    //app-specific code to delete the specified booking ID
```

Where,

- /travel-agent/api/bookings/reserve, /travel-agent/api/confirm/<trip_booking_id>, and /travel-agent/api/cancel/<trip_booking_id> are the REST API endpoints that the transaction initiator service exposes. Ensure that these endpoints are present in the transaction initiator service and the confirm and cancel logic is implemented in the code.
- do_trip_reserve(), do_trip_confirm(), and do_trip_cancel() methods contain the business logic for creating a reservation, confirming a reservation, canceling a reservation respectively. Ensure that the business logic is implemented in the code of the transaction initiator service and the endpoints are present.

Source code of a sample Python transaction initiator application which uses the MicroTx Distributed Transactions library is available in the [app.py](#) file which is located in the microtx-samples GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

5.8.2 Configure Python App as Transaction Participant

You can select Flask or Django as the framework for your Python application. This section provides instructions to integrate the MicroTx Distributed Transactions library with the application code of your Python application with Flask framework.

1. Open a terminal in the virtual environment that you have created for your Python application, and then run the following command to install the MicroTx Distributed Transactions library file for Python which is available in the installation_directory/MicroTx-<version>/lib/python folder.

```
pip3 install tmm.py-<version>.whl
```

2. Configure the property values for the MicroTx Distributed Transactions library. Create a new file and save it as tmm.properties. You must provide values for the following properties.

The following example provides sample values for the properties. Provide the values based on your environment.

```
oracle.tmm.PropagateTraceHeaders = true
server.port = 8080
oracle.tmm.CallbackUrl = http://localhost:{server.port}
```

For details about each property, see [Configure Library Properties](#).

Note down the name of this file as you will have to provide this later.

3. Import the MicroTx Distributed Transactions libraries and exceptions. You can use `tcclib.exception` to handle exceptions.

```
from tcclib.tcc import TCCClient, Middleware, http_request, TCCConfig
import tcclib.exception as ex
```

4. Create a Flask application and a middleware object.

The following sample code creates a Flask application named `app` and a middleware object. The middleware object wraps around the Flask application and intercepts all the incoming requests received by the Flask application.

```
# Create an instance of the Flask class with the name of the current
module.
app = Flask(__name__)
# Create a middleware object to wrap around the Flask application that you
have created.
# The middleware object intercepts all the incoming requests received by
the Flask application.
app.wsgi_app = middleware(app.wsgi_app)
```

5. Add the following code to initialize the `tccConfig` object for the microservice.

Syntax

```
tccConfig = TCCConfig(filePath=<application_properties_file_path>,
timeLimitInSeconds=<integer>)
```

Sample

```
tccConfig = TCCConfig(filePath="./tmm.properties", timeLimitInSeconds=300)
```

Where,

- `./tmm.properties` is the location of the file in which you have defined values for the MicroTx Distributed Transactions library properties for the transaction participant service.
- `300` is the time limit in seconds for the transaction participant service to reserve the resources. Specify the time period as a whole number. It is the responsibility of the application developer to provide the required code to release the resources and cancel their part of the TCC transaction after the time limit expires. Decide the time limit based on your business requirement.

Replace these values with the values specific to your environment.

6. Use the TCCConfig object that you have created earlier to register participants (reserved resource URI) to an existing TCC transaction by calling the addTccParticipant method with the resource URI.

```
const bookingUri;  
tccConfig.addTccParticipant(bookingUri);
```

When this code is executed, the participant service joins an existing TCC transaction when the initiator service calls the participant service. Also the MicroTx Distributed Transactions library enlists the participant service with the URIs you provide for the confirm and cancel endpoints.

Source code of a sample Python transaction participant application which uses the MicroTx Distributed Transactions library is available in the [app.py](#) file which is located in the microtx-samples GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

6

Develop Tuxedo Apps with XA

Enable Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions) interoperability in your Tuxedo or SALT applications. You can only use the XA transaction protocol for your Tuxedo or SALT applications.

- [Run Tuxedo App on Linux Host](#)
- [Run Tuxedo App in Kubernetes Cluster](#)

6.1 Run Tuxedo App on Linux Host

The environment contains at least two hosts:

- The Linux host in which the Tuxedo application runs.
- The physical or virtual host on which you have deployed the Kubernetes cluster in which you have installed MicroTx Distributed Transactions.
- [Prepare the Environment](#)
Ensure that network connectivity is there between the Tuxedo application and MicroTx Distributed Transactions.
- [Install Patches](#)
Apply patches for Tuxedo and SALT on your Tuxedo host.
- [Verify the Set Up](#)
To verify that you have set up the Tuxedo environment properly, download the sample Tuxedo application and run it in the Tuxedo environment.

6.1.1 Prepare the Environment

Ensure that network connectivity is there between the Tuxedo application and MicroTx Distributed Transactions.

Before you begin, complete the following tasks:

- Identify the Tuxedo environment that you want to use. You can use an existing Tuxedo environment or create a new one.
 - To create a new Tuxedo environment, install Tuxedo 12cR2 (12.2.2) on a 64-bit Linux server. The default installation options also installs SALT, which is needed for MicroTx Distributed Transactions interoperability. For information about the Linux platforms that Tuxedo supports, see https://docs.oracle.com/cd/E72452_01/tuxedo/docs1222/install/inspds.html.
You can download the installer from <https://www.oracle.com/middleware/technologies/tuxedo-downloads.html>. For details about the installation steps, see https://docs.oracle.com/cd/E72452_01/tuxedo/docs1222/install/.
Skip this step if you are using an existing Tuxedo environment.
- Install MicroTx Distributed Transactions.

To verify bi-directional network connection between the Tuxedo application and MicroTx Distributed Transactions:

1. Use SSH to login to the MicroTx Distributed Transactions host and the Linux host on which you have installed the Tuxedo application.
2. Start a simple HTTP server on the MicroTx Distributed Transactions host.

```
python -m SimpleHTTPServer 2345
```

Where, 2345 is a port in the MicroTx Distributed Transactions host for which you have set up the required networking rules to permit traffic.

Note down the host name of the HTTP server that is created. You will provide this information in the next step.

3. On the Tuxedo host, run the following command to connect to the HTTP server that is running on the MicroTx Distributed Transactions host.

Command Syntax

```
curl -vv Transaction_Manager_for_Microservices_host_name:2345
```

Where,

- *Transaction_Manager_for_Microservices_host_name* is the name of the physical or virtual host, of the Kubernetes cluster or Docker container, on which you have installed MicroTx Distributed Transactions.
 - 2345 is a port in the MicroTx Distributed Transactions host for which you have set up the required networking rules to permit traffic.
4. Start a simple HTTP server on the Tuxedo host.

```
python -m SimpleHTTPServer 2345
```

Where, 2345 is a port in the Tuxedo host for which you have set up the required networking rules to permit traffic.

Note down the host name of the HTTP server that is created. You will provide this information in the next step.

5. On the MicroTx Distributed Transactions host, run the following command to connect to the HTTP server that is running on the Tuxedo host.

Command Syntax

```
curl -vv Tuxedo_host_name:2345
```

If the command is not successfully executed, it indicates that there is a networking problem between the two hosts. Troubleshoot the networking issue. For example, you may need to open the ports to permit traffic.

6.1.2 Install Patches

Apply patches for Tuxedo and SALT on your Tuxedo host.

To apply the Tuxedo and SALT patches:

1. Shutdown your Tuxedo application before applying a patch.
2. To apply the Tuxedo patch:
 - a. Download the patch 24574032. For details about downloading the patch, see [Downloading Release Update Patches](#) in *Database Client Installation Guide for Linux*.

- b. Unzip the patch file.

```
unzip p33664689_122200_Linux-x86-64.zip
```

- c. Apply the patch.

Command syntax

```
cd $ORACLE_HOME/OPatch  
./opatch apply fullpath_of_the_patch_file
```

Sample command

```
cd $ORACLE_HOME/OPatch/  
./opatch apply 33664689.zip
```

3. To apply the SALT patch:
 - a. Ensure that your Tuxedo application is still shutdown.
 - b. Unzip the RP902.zip file that was supplied with the sample files.

```
unzip RP902.zip
```

- c. Apply the patch.

Command syntax

```
cd $ORACLE_HOME/OPatch  
./opatch apply fullpath_of_the_patch_file
```

Sample command

```
cd $ORACLE_HOME/OPatch/  
./opatch apply 99999999.zip
```

4. Set the environment variable. For information about other Tuxedo environment variables that you can set, see [Setting Environment Variables](#).

Syntax

```
export SALT_TMM_CALLBACK_ADDR=http://IP_Address:GWWS Port
```

Where,

- *IP_Address*: Enter the IP address of the host on which the GWWS server is running.
- *GWWS Port*: You can find this value in the bankapp.dep file. The default value is 2345. If you change the GWWS port, update the value in the bankapp.dep file as well.

Example

```
export SALT_TMM_CALLBACK_ADDR=http://192.0.2.6:2345
```

5. Run the following command to verify the patch installation.

```
wsadmin -v
```

The following information is displayed which verifies that the patch has been applied.

```
INFO: Oracle SALT, Version 12.2.2.0.0, 64-bit, Patch Level 902  
INFO: Oracle Tuxedo, Version 12.2.2.0.0, 64-bit, Patch Level 086
```

6. Start the Tuxedo application.

6.1.3 Verify the Set Up

To verify that you have set up the Tuxedo environment properly, download the sample Tuxedo application and run it in the Tuxedo environment.

Before you run an XA transaction using MicroTx Distributed Transactions, you must run the sample Tuxedo application in your Tuxedo environment to ensure that you have set up the environment properly.

1. Download the sample code and application binaries (.zip file) from the link provided to you by the Product team.
2. Unzip the sample code bundle in the parent directory of \$TUXDIR.

```
unzip bankapp-env.zip
```

The following new files and folders are available: Dockerfile file, install1222.rsp file, start.sh file, and bankapp folder.

3. Initialize the Tuxedo environment variables.

```
cd parent_directory_of_$TUXDIR  
. tuxedo12.2.2.0.0/tux.env
```

This also sets the value for the TUXDIR environment variable.

4. Navigate to the bankapp folder.

```
cd bankapp
```

5. Run the bankvar file to set up the environment variables for the Tuxedo sample application.

```
./bankvar
```

This also sets the value for the APPDIR environment variable.

- Run the following script to update the settings in different files.

```
now=$(date +%m%d%H%M%S)
for f in "bankapp.dep" "bankapp.mk" "bankvar" "ENVFILE" "TMUSREVT.ENV"
"ubbsbm"
do
    cp $f $f.${now}
    test -e $TUXDIR && test -e $APPDIR && sed -i -e "s^/u01/data/bankapp^$
{APPDIR}^" -e "s^/u01/app/tuxedo12.2.2.0.0^${TUXDIR}^" $f
done
```

Set the two environment variables TUXDIR and APPDIR according to your environment. Set the value for APPDIR to point to the directory that contains the files for the banking application.

- Run the following commands to rebuild and start the sample Tuxedo application.

```
rm -f TLOG GWTLOG tuxconfig saltconfig bankdl1 bankdl2 bankdl3
. ./bankvar
tmloadcf -y ubbsbm
wsloadcf -y bankapp.dep
./crbank
./crtlog
tmboot -y
./populate
```

- Run the following commands to verify that the sample Tuxedo application works and returns the expected response.

```
# Query account, readonly operation
curl -X POST -H "Content-type:application/json" http://
Tuxedo_host_name_or_IP_address:2345/INQUIRY -d '{"ACCOUNT_ID":10001}'

# Withdrawal API
curl -X POST -H "Content-type:application/json" http://
Tuxedo_host_name_or_IP_address:2345/WITHDRAWAL -d
'{"ACCOUNT_ID":10001,"SAMOUNT":"1"}'

# Deposit API
curl -X POST -H "Content-type:application/json" http://
Tuxedo_host_name_or_IP_address:2345/DEPOSIT -d
'{"ACCOUNT_ID":10001,"SAMOUNT":"1"}'

# Transfer API
curl -X POST -H "Content-type:application/json" http://
Tuxedo_host_name_or_IP_address:2345/TRANSFER -d '{"ACCOUNT_ID":
[10001,10002],"SAMOUNT":"1"}'
```

Where, *Tuxedo_host_name_or_IP_address* is the IP address or the name of the host on which you have installed Tuxedo. Run `uname -n` to find the name of the host. Run `ifconfig` to get the IP address of the host.

You can change the value of the port and the ACCOUNT_ID based on your environment.

For each Tuxedo application that you want to use with MicroTx Distributed Transactions, create separate configuration files and use separate ports.

After your Tuxedo application is running, make changes to the initiator application in the sample XA application. In the initiator application, configure the Tuxedo application as a participant application. When you run an XA transaction using the sample app, the initiator application sends requests to your Tuxedo participant application.

After installing the sample application, run an XA transaction.

After you successfully install and run the sample application, your environment is ready for you to create and run your own Tuxedo applications.

6.2 Run Tuxedo App in Kubernetes Cluster

Run the Tuxedo application in the same Kubernetes cluster in which you have deployed MicroTx Distributed Transactions.

- [Start Tuxedo Sample App in a Docker Container](#)
- [Update the YAML Files for Tuxedo App](#)
The sample application files also contain the `sampleapps.yaml` and `values.yaml` file. Provide details about your Tuxedo application in these YAML files.

6.2.1 Start Tuxedo Sample App in a Docker Container

Before you begin, download the sample code and application binaries (.zip file) from the link provided to you by the Product team. The sample Tuxedo application is a banking application.

1. Build Docker images for your Tuxedo and SALT application.

```
cd parent_directory_of_$TUXDIR
docker build -t tmmbankapp_sample .
```

If you don't want to build a docker image for the Tuxedo sample application, contact the Product team to get the Docker image and sample code.

2. Start the application in the Docker container.

```
docker run -d --env SALT_TMM_CALLBACK_ADDR=http://Tuxedo_hostname:2345 -p
2345:2345 tmmbankapp_sample
```

Wait for about 15 seconds, until the Tuxedo application boots fully.

Ensure that the sample application is working correctly in the Tuxedo environment. See [Verify the Set Up](#).

6.2.2 Update the YAML Files for Tuxedo App

The sample application files also contain the `sampleapps.yaml` and `values.yaml` file. Provide details about your Tuxedo application in these YAML files.

To provide the configuration and image details about the Tuxedo app in the YAML files:

1. Back up the `sampleapps.yaml` file, which is located in the `xa/java/helmcharts/transfer/templates` folder in the [microtx-samples](#) GitHub repository, in a different location outside the templates folder.
2. Open the `sampleapps.yaml` file in any code editor to edit it.

3. Replace the dept2 service deployment descriptor with the sample code provided below.

```

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dept2
  labels:
    app: dept2
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dept2
      version: v1
  template:
    metadata:
      labels:
        app: dept2
        version: v1
    spec:
      containers:
        - name: dept2
          image: #TUXEDO-BANK-APP-IMAGE
          imagePullPolicy: Always
          ports:
            - containerPort: 2345
          env:
            - name: SALT_TMM_CALLBACK_ADDR
              value: http://dept2:2345
      imagePullSecrets:
        - name: regcred
---

```

Where, you can replace the following details with values specific to your environment.

- #TUXEDO-BANK-APP-IMAGE: Provide details of the Tuxedo application image that you have uploaded to the docker container. For example, iad.ocir.io/mytenancy/xa/tuxedo-app-xa:v1.
 - 2345 is a port in the Tuxedo host for which you have set up the required networking rules to permit traffic.
4. In the deployment descriptor for the accounts service, modify the value for the departmentTwoEndpoint environment variable to the value you have specified for the SALT_TMM_CALLBACK_ADDR environment variable.

```

name: departmentTwoEndpoint
value: http://dept2:2345

```

5. Save your changes.
6. Back up the values.yaml file, which is located in the *installation_directory*/MicroTx-RELEASE/samples/xa/helmcharts/transfer folder, in a different location outside the transfer folder.

7. Replace the values for dept2 with the content provided below.

```
dept2:
  name: dept2
  host: dept2
  version: v1
  gatewayUriPrefix: /dept2/
  rewriteUriPrefix: /
  destinationHost: dept2
```

8. Save your changes.

After modifying the `sampleapps.yaml` and `values.yaml` files, install the XA sample application using Helm. Helm deploys the Tuxedo bank app container in the Kubernetes cluster along with the other sample apps for XA. After installing the sample application, run an XA transaction. For information about installing and running the sample application, see the `readme.md` file located in the `xa` folder in the [microtx-samples](#) GitHub repository.

After you successfully install and run the sample application, your environment is ready for you to create and run your own Tuxedo applications.

Manage the Transaction Coordinator

If you have installed Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions) on a Kubernetes cluster using Helm, you can use Helm commands to manage the transaction coordinator.

- [About Session Affinity](#)
When you enable session affinity, all the requests for a unique transaction or session are routed to the same endpoint or replica of the participant service that served the first request.
- [Enable Caching](#)
When you enable caching, it optimizes the read and write operations for the transaction logs that are stored in etcd or Oracle Database. This in turn improves the performance.
- [About Transaction Recovery](#)
The transaction coordinator server resumes the transactions that were in progress when the server restarts after a failure.
- [General Syntax of Commands](#)
The following is the general syntax of the Helm commands that you can run to manage MicroTx Distributed Transactions.
- [Scale up or down](#)
Scale up the Kubernetes cluster on which you have installed MicroTx Distributed Transactions to handle a large amount of requests. When the number of requests is low, scale down to use the resources efficiently.
- [Update](#)
- [Uninstall](#)

7.1 About Session Affinity

When you enable session affinity, all the requests for a unique transaction or session are routed to the same endpoint or replica of the participant service that served the first request.

Use a sticky session to associate a service instance, a Kubernetes pod or a replica, with an application based on the `oracle-tmm-txn-id` HTTP header. A consistent hash is created based on the `oracle-tmm-txn-id` HTTP header, and then the sticky session is established. The MicroTx Distributed Transactions library and transaction coordinator include the `oracle-tmm-txn-id` HTTP header in all subsequent calls.

When the transaction initiator service calls the participant service, the MicroTx Distributed Transactions library injects the `oracle-tmm-txn-id` HTTP header in the outgoing request. All subsequent calls from MicroTx Distributed Transactions to the participant service also include this header. In this manner all requests are routed to the concerned replica of the transaction participant service.

Based on your business use case, you will need to enable session affinity for a participant service or for the transaction coordinator. When there are multiple replicas of a participant service, the request may be directed to different replicas in a single transaction. When you enable session affinity for a participant service, all the requests for a particular transaction or

session are routed to the same endpoint or replica of the participant service that served the first request.

If you enable session affinity when it isn't required, it may have an adverse impact on the application's performance.

When should you enable session affinity for an XA participant service

Enable session affinity for an XA participant service in the following scenarios only if there are multiple instances or replicas of the participant service, so that all requests are routed to the concerned replica. You must enable session affinity or sticky sessions for an XA participant service in the following scenarios.

- When a transaction participant uses a non-XA resource and the Logging Last Resource (LLR) or Last Record Commit (LRC) optimization is enabled.
- When a transaction participant uses PostgreSQL as a resource manager, that requires you to use the same session for initiating the XA transaction and for all subsequent requests.

When should you enable session affinity for a Saga participant service

Enable session affinity for a Saga participant service if there are multiple instances or replicas of the participant service, so that all requests are routed to the concerned replica.

When should you enable session affinity for transaction coordinator

You don't need to enable session affinity for TCC transactions. For Saga and XA transactions, you must enable session affinity for the transaction coordinator in the following scenarios:

- When you use internal memory as data store and deploy the transaction coordinator on more than one replica. This ensures that all requests are routed to the concerned replica of the transaction coordinator.
- When you enable caching.

For information about enabling session affinity for a participant service or transaction coordinator, see [Enable Session Affinity](#).

For each participant service, you may run one or more replicas of the service. The session affinity to a particular host is lost when you add or remove replicas for a participant service. For more details, see <https://istio.io/latest/docs/reference/config/networking/destination-rule/#LoadBalancerSettings-ConsistentHashLB>.

- [Enable Session Affinity](#)
When you enable session affinity, all the requests for a unique transaction or session are routed to the same endpoint or replica of the participant service that served the first request.

7.1.1 Enable Session Affinity

When you enable session affinity, all the requests for a unique transaction or session are routed to the same endpoint or replica of the participant service that served the first request.

Use the instructions provided in this section to enable session affinity or sticky sessions if you have deployed the participant service or transaction coordinator within an Istio service mesh. If you are using load balancers, such as Nginx or HAProxy, refer to the load balancer documentation to configure session affinity or sticky sessions.

The steps provided in this section are specific to enabling session affinity for a participant service. You can enable session affinity for the transaction coordinator in a similar manner.

Before you begin, complete the following tasks:

1. Ensure that you have deployed the transaction coordinator and participant services within an Istio service mesh.
2. Identify if you need to enable session affinity for your participant service or for the transaction coordinator. See [About Session Affinity](#).

To enable session affinity for a participant service:

1. Enable session affinity for the MicroTx Distributed Transactions transaction coordinator. In the values.yaml file of the MicroTx Distributed Transactions transaction coordinator, add the following line of code. See [Configure the values.yaml File](#).

```
sessionAffinity: true
```

2. Enable session affinity for transaction participant services and the transaction initiator service. In the values.yaml file, which is located in the Helm Chart of the service, add the following line of code.

```
sessionAffinity: true
```

Note

If your application uses the Saga transaction protocol, you must set the sessionAffinity property to true in all the participant services and the transaction initiator service. In case of XA transaction protocol, you can set the sessionAffinity property to true in one or more participant services.

7.2 Enable Caching

When you enable caching, it optimizes the read and write operations for the transaction logs that are stored in etcd or Oracle Database. This in turn improves the performance.

You can not enable caching if you use internal memory as the transaction store.

To enable caching, perform the following tasks:

1. Open the values.yaml file in any code editor. This file is located in the installation_directory\otmm-RELEASE\otmm\helmcharts folder.
2. In the YAML file, set tmmConfiguration.caching.enabled to true and specify the maximum storage capacity of the cache in GB. The following code snippet provides sample values for the caching property in the values.yaml file.

```
caching:  
  enabled: true  
  maxCapacity: 1.5
```

3. Save your changes.
4. Enable session affinity. See [Enable Session Affinity](#).

7.3 About Transaction Recovery

The transaction coordinator server resumes the transactions that were in progress when the server restarts after a failure.

Every time the transaction coordinator server restarts, it goes through all the in-progress transactions stored in the transaction store and restarts the ongoing transactions. The recovery depends on the data that is available in the transaction store. The transaction store should retain information about the earlier transactions even after the transaction coordinator crashes or restarts. If you have set up etcd or Oracle Database for MicroTx Distributed Transactions to store the transaction data, then you can obtain information about the in-progress transactions and transaction details after the coordinator restarts. However, if you haven't set up a separate transaction store and are using internal memory to store the transaction details, then all the stored information is lost after the coordinator crashes or restarts.

MicroTx Distributed Transactions recovers in-progress transactions, based on the data available in the transaction store, for transactions which are in the following states:

Transaction protocol	Transaction status	As part of transaction recovery, the transaction coordinator...
XA	Preparing	rolls back the transactions
XA	Rolling back	rolls back the transactions
XA	Committing	resends the prepare and commit commands and retries to commit the transactions successfully
Saga	Closing	reissues the close command to close the transaction
Saga	Canceling	reissues the cancel command to cancel the transaction
TCC	Confirming	reissues the confirm command to confirm the transaction
TCC	Canceling	reissues the cancel command to cancel the transaction

Additionally, for XA transaction protocol, the transaction coordinator dynamically recovers the transactions which are not committed. See [About Dynamic Recovery for XA Transactions](#).

7.4 General Syntax of Commands

The following is the general syntax of the Helm commands that you can run to manage MicroTx Distributed Transactions.

```
helm action release_name --namespace namespace --reuse-values --values  
file.yaml chart_directory
```

General Command Actions

The following table describes the general actions that you can perform to manage MicroTx Distributed Transactions.

Action	Description
upgrade	Updates one or more values that you have defined for MicroTx Distributed Transactions in its <code>values.yaml</code> file.
uninstall	Removes MicroTx Distributed Transactions from the Kubernetes cluster.

Command Parameter

Parameter	Description
release_name	Enter the name of the MicroTx Distributed Transactions application on which you want to perform an action, such as update. You provided this name while installing the application.

Command Options

The following table describes the general actions that you can perform on Transaction Coordinating Server.

Option	Description
namespace	Enter the Kubernetes namespace where you have deployed MicroTx Distributed Transactions. Example, MicroTx.
reuse-values	Specify this option to modify only those values which you specify in the YAML file while retaining all the other values as is.
values	Enter the name of the YAML file that you have created which contains the values that you want to modify. Example, file_updated_values.yaml.
chart_directory	Specify the location of the folder that contains the chart.yaml file for the MicroTx Distributed Transactions application. Example, installation_directory\MicroTx-RELEASE\MicroTx\helmcharts.

7.5 Scale up or down

Scale up the Kubernetes cluster on which you have installed MicroTx Distributed Transactions to handle a large amount of requests. When the number of requests is low, scale down to use the resources efficiently.

The replica count is the number of replicas of the MicroTx Distributed Transactions instance that you want to run at a time.

Perform the following steps to scale the Kubernetes cluster on which you have installed MicroTx Distributed Transactions:

1. In any text editor, create a YAML file with updated replicaCount value. In the following example, the replicaCount value is mentioned as 3.


```
tmmReplicaCount: 3
```
2. Validate and save the YAML file.
3. Run the following command to upgrade the Kubernetes cluster on which you have installed MicroTx Distributed Transactions based on the replicaCount value provided in the scale.yaml file.

Syntax

```
helm upgrade <release name> --namespace <namespace> --reuse-values --values  
scale.yaml <chart directory>
```

For information about the general command parameter and command options, see [General Syntax of Commands](#).

Example

The following command scales the Kubernetes cluster in the MicroTx namespace with the details mentioned in the `scale.yaml` file.

```
helm upgrade MicroTx --namespace MicroTx --reuse-values --values scale.yaml  
MicroTx-RELEASE\MicroTx\helmcharts\
```

Usage Notes

When you run this command, the Kubernetes cluster is not recreated. Based on the replica count that you specify, Kubernetes starts new replicas or stops existing replicas to match the specified replica count.

Helm performs rolling upgrade. Old replicas are gradually removed, while new replicas are started. Traffic is gradually shifted to the new replicas from old replicas and the old replicas are terminated only when all the traffic has been shifted to the new replicas. This ensures that there is no loss of in flight transactions.

7.6 Update

Use this command to update one or more property values that you have defined for MicroTx Distributed Transactions in its `values.yaml` file. You can update the properties for the authorization, authentication, transaction store, encryption key, and transaction token, transaction time out, and other details.

1. In any text editor, create a YAML file. Specify the values that you want to update in the YAML file.
The following code sample shows the updated value for logging level.

```
logging:  
  level: warning
```

2. Run the following command to update the properties of the Kubernetes cluster on which you have installed MicroTx Distributed Transactions.

Syntax

```
helm upgrade <release name> --namespace <namespace> --reuse-values --  
values <file_name.yaml> <chart directory>
```

Specify the `reuse-values` option to modify only those values that you specify in the YAML file, while retaining all the other values as is. For information about the general command parameter and command options, see [General Syntax of Commands](#).

Example

The following command updates the Kubernetes cluster in the `omtm` namespace with the details mentioned in the `update_log_level.yaml` file.

```
helm upgrade omtm --namespace omtm --reuse-values --values  
update_log_level.yaml omtm/
```

Usage Notes

Helm performs rolling upgrade. Old replicas are gradually removed, while new replicas are started. Traffic is gradually shifted to the new replicas from old replicas and the old replicas are terminated only when all the traffic has been shifted to the new replicas. This ensures that there is no loss of in flight transactions.

7.7 Uninstall

When you no longer want to use MicroTx Distributed Transactions, you can uninstall it from the Kubernetes cluster.

Prerequisites

Before you run this command, ensure that you do not need to run MicroTx Distributed Transactions in your deployment and that there are no active transactions in progress.

Syntax

```
helm uninstall release_name
```

Where,

release_name is the name of the application that you want to uninstall.

Usage Notes

When you run this command, it removes MicroTx Distributed Transactions from the Kubernetes cluster. Communication between your applications microservices will continue, but any requests sent by the application microservices to MicroTx Distributed Transactions will fail.

Examples

Use the following command to delete the `tmm_app` application.

```
helm uninstall tmm_app
```

8

Manage Transactions Using the Web Console

Use the MicroTx Distributed Transactions console to view and manage transactions for all the replicas of transaction coordinator in a Kubernetes cluster.

A Kubernetes cluster may contain one or more replicas of the transaction coordinator. The transaction statistics are displayed for all instances of the transaction coordinator and not for a specific replica.

- [Access the Web Console](#)
You can manage and monitor the transactions using an easy-to-use graphical web console.
- [View Details of a Transaction](#)
The **Transactions** tab displays transactions that MicroTx Distributed Transactions is currently processing. These transactions could be in various states, including the heuristic state.
- [View the Total Number of Transactions](#)
You can view the total number of transaction requests that MicroTx Distributed Transactions has received since you had installed it and the number of transactions in different states.
- [View the Number of Recent Transactions](#)
View the number of transactions and their state in a specified time interval.
- [View Coordinator Health](#)
View the number of replicas of MicroTx Distributed Transactions that are in the running state and the percentage of CPU and memory that each replica currently consumes.
- [View Console Health](#)
- [View Configuration Properties](#)
You can view the property values that you have set for MicroTx Distributed Transactions transaction coordinator in the `values.yaml` file, but you can't edit it using the MicroTx Distributed Transactions console.
- [Manage XA Transactions](#)
You can commit or roll back an active XA transaction. You can delete an XA transaction which is in the heuristic state.
- [Manage TCC Transactions](#)
- [Manage Saga Transactions](#)

8.1 Access the Web Console

You can manage and monitor the transactions using an easy-to-use graphical web console.

To provide access to the console, use the identity provider that you have set up before installing MicroTx Distributed Transactions.

Note

Administrators can access the MicroTx Distributed Transactions console to view and manage transactions by all the users. Non-admin users can only view and manage transactions that they have initiated. They can not view or manage transactions of other users. To set permissions for administrators, see Console Configuration Properties.

To access the MicroTx Distributed Transactions console:

1. Create the URL in the following format.

Syntax of URL

`https://externalHostname/consoleui/`

Where, *externalHostname*: The IP address of the load balancer of the Istio ingress gateway. See [Find IP Address of Istio Ingress Gateway](#). For example, 192.0.2.1.

Based on the example values provided above, the example URL to access the MicroTx Distributed Transactions console is `https://192.0.2.1/consoleui/`.

2. Enter the MicroTx Distributed Transactions console URL in any browser.

The MicroTx Distributed Transactions console works with browsers that support OJET. For the list of supported browsers, see <https://www.oracle.com/webfolder/technetwork/jet-900/globalSupport-FAQ.html#platforms>.

3. Enter the user name and password.

8.2 View Details of a Transaction

The **Transactions** tab displays transactions that MicroTx Distributed Transactions is currently processing. These transactions could be in various states, including the heuristic state.

The details of transactions which have reached their final state, such as confirmed, canceled, rolled back, or committed are deleted after the time interval specified in the `completedTransactionTTL` property in the `values.yaml` file. When the details of a transaction is deleted, it will no longer be displayed on the **Transactions** tab.

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Transactions** tab.
3. In the **Transaction Type** drop-down list, select the transaction protocol for which you want to view details, such as the **XA**, **Saga**, or **TCC**.
4. Select the transaction for which you want to view details.
5. Click **Action** , and then click **Details**.

The following details are displayed.

Transaction Detail	Description
Transaction ID	The unique ID of the transaction. You can use this ID to track the progress of the transaction.
Status	The current status of the transaction.

Transaction Detail	Description
Begin Time	The time at which the transaction was started. This is listed only for the transactions that use the Saga and XA transaction protocols.
Remaining Time	The estimated time remaining for the transaction to reach a final state.
End Time	The time at which the transaction reached a final state.
Transaction Type	The transaction protocol, such as the XA , Saga , or TCC .
Participants	Status of the participant services and their URIs for transactions that use the Saga or TCC transaction protocol. This helps to determine the state of every participant service.
Branch ID	Status of the participant services and their URL for transactions that use the XA transaction protocol. This helps to determine the state of every participant service. For example, if a transaction is rolled back, the information in this section can help you identify which participant service caused the roll back. Only in case of the XA transaction protocol, MicroTx Distributed Transactions creates a unique branch ID when each participant service enlists to MicroTx Distributed Transactions. For example, when the participant service enlists, it provides <code>http://dept1:8081</code> as the URL to the MicroTx Distributed Transactions coordinator. When a service connects to multiple resource managers, MicroTx Distributed Transactions creates a unique branch ID for each resource manager.

The following image shows the details of a TCC transaction which is in heuristic state. The **Participants** section provides details about the participant service that failed to confirm the TCC transaction.

The screenshot displays the Oracle Transaction Manager for Microservices interface. The left sidebar shows the 'Transactions' tab selected. The main area is divided into two sections: 'Transaction Details' and 'Details'.

Transaction Details Table:

Transaction Id	End Time
8ece58b6-7308-4e08-89f0-e29214963ae3	18-Jun-2024 18:28:12 GMT+05:30
bcb6152-9500-466f-b49e-31b2e5a5cfe4	18-Jun-2024 18:36:43 GMT+05:30
e50e1bf9-4976-4d57-aa40-cf9f06f7872b	1-Jul-2024 14:21:6 GMT+05:30
98bbe331-084d-473c-e5e2-a0a5c708c570	1-Jul-2024 14:29:50 GMT+05:30

Details Panel:

- Transaction Id:** 8ece58b6-7308-4e08-89f0-e29214963ae3
- Status:** Heuristic
- End Time:** 18-Jun-2024 18:28:12 GMT+05:30
- Transaction Type:** TCC
- Participants:**

Status	URI
Confirmed	http://hotel-booking:8081/api/bookings/5259f9d0-e312-4a8d-8f
FailedToConfirm	http://flight-booking:8082/api/bookings/71361f4d-e430-432f-ae

8.3 View the Total Number of Transactions

You can view the total number of transaction requests that MicroTx Distributed Transactions has received since you had installed it and the number of transactions in different states.

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).

2. Click the **Dashboard** tab.
3. In the **Refresh** drop-down list, select the rate at which you want the data on the console to be refreshed. If you select 30 seconds, data for coordinator health, console health, and transaction summary is updated every 30 seconds on the console.
4. Under **Transaction Summary**, view the number of transactions in different states since MicroTx Distributed Transactions was installed.
 - **Processed** displays the total number of transactions that MicroTx Distributed Transactions has processed irrespective of their final state, such as committed, cancelled, or heuristic. It lists the total number of transaction requests that were processed from the time MicroTx Distributed Transactions was installed.
 - **Confirmed** displays the total number of committed transactions for XA and TCC and confirmed transactions for Saga.
 - **Cancelled** displays the total number of transactions that were rolled back for XA and the total number of transactions that were canceled for Saga and TCC.
 - **Heuristic**: displays the total number of transactions in the heuristic state for XA; transactions in failed to cancel and failed to close states for Saga; and failed to cancel, failed to confirm, and heuristic states for TCC.

8.4 View the Number of Recent Transactions

View the number of transactions and their state in a specified time interval.

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Dashboard** tab.
3. In the **Refresh** drop-down list, select the rate at which you want the data on the console to be refreshed. If you select 30 seconds, data for coordinator health, console health, and transaction summary is updated every 30 seconds on the console.
4. In the **Interval** box, enter the time period in seconds for which you want to view the most recent transactions and active transactions. For the time interval, the minimum value is 120 seconds and the maximum value is 1800 seconds.

The **Most Recent Transactions** and **Active Transactions** graph displays the number of transactions and their state in the specified time interval. For example, if you enter 300, then **Most Recent Transactions** displays the total number of transactions processed in the last 300 seconds along with their latest state. **Active Transactions** displays the number of transactions that the MicroTx Distributed Transactions is currently processing.

8.5 View Coordinator Health

View the number of replicas of MicroTx Distributed Transactions that are in the running state and the percentage of CPU and memory that each replica currently consumes.

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Dashboard** tab.
3. In the **Refresh** drop-down list, select the rate at which you want the data on the console to be refreshed. If you select 30 seconds, data for coordinator health, console health, and transaction summary is updated every 30 seconds on the console.
4. In the **Coordinator Health** section, click **CPU** or **Memory** to view the percentage of CPU or memory consumed at the current moment.

8.6 View Console Health

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Dashboard** tab.
3. In the **Refresh** drop-down list, select the rate at which you want the data on the console to be refreshed. If you select 30 seconds, data for coordinator health, console health, and transaction summary is updated every 30 seconds on the console.
4. In the **Console Health** graph, click **CPU** or **Memory** to view the percentage of the available CPU or memory that has been consumed.

8.7 View Configuration Properties

You can view the property values that you have set for MicroTx Distributed Transactions transaction coordinator in the `values.yaml` file, but you can't edit it using the MicroTx Distributed Transactions console.

To update the property values of the transaction coordinator, modify the property values in the `values.yaml` file, and then run the `helm upgrade` command. See [Update](#).

To view the property values of the transaction coordinator:

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Coordinator Configuration** tab.
3. View the property values that you have set in the `values.yaml` file for the transaction coordinator.

To view the description for each property, see MicroTx Distributed Transactions Coordinator Configuration.

8.8 Manage XA Transactions

You can commit or roll back an active XA transaction. You can delete an XA transaction which is in the heuristic state.

If you can't roll back, commit, or delete an XA transaction, then the options are grayed out so you won't be able to perform these operations.

- [Commit an XA Transaction](#)
You can commit an active XA transaction.
- [Roll Back an XA Transaction](#)
You can roll back an active XA transaction.
- [Delete an XA Transaction](#)
You can delete an XA transaction that is in the `HeuristicallyCompleted` state.

8.8.1 Commit an XA Transaction

You can commit an active XA transaction.

Within the XA transaction, all requests to participant services either succeed or they all are rolled back in case of a failure of any one or more actions. The transaction initiator application commits the XA transaction if all service requests are executed successfully.

To commit an active XA transaction:

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Transactions** tab.
3. In the **Transaction Type** drop-down list, select **XA** to view the transaction details for applications using the XA transaction protocol.
4. Click **Action** , and then click **Commit**.

8.8.2 Roll Back an XA Transaction

You can roll back an active XA transaction.

Within the XA transaction, all requests to participant services either succeed or they all are rolled back in case of a failure of any one or more actions. The transaction initiator application commits the XA transaction if all service requests are executed successfully.

To roll back an active XA transaction:

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Transactions** tab.
3. In the **Transaction Type** drop-down list, select **XA** to view the transaction details for applications using the XA transaction protocol.
4. Click **Action** , and then click **Roll Back**.

8.8.3 Delete an XA Transaction

You can delete an XA transaction that is in the `HeuristicallyCompleted` state.

To delete an XA transaction:

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Transactions** tab.
3. In the **Transaction Type** drop-down list, select **XA** to view the transaction details for applications using the XA transaction protocol.
4. Click **Action** , and then click **Delete**.

8.9 Manage TCC Transactions

- [Confirm a TCC Transaction](#)
Confirms the resources that are held in a reserved state in the TCC transaction. To confirm the transaction, the transaction initiator service calls the transaction coordinator, which then calls each participant service.
- [Cancel a TCC Transaction](#)
Cancels the resources that are held in a reserved state in the TCC transaction. To cancel the transaction, the transaction initiator service calls the transaction coordinator, which then calls each participant service.

8.9.1 Confirm a TCC Transaction

Confirms the resources that are held in a reserved state in the TCC transaction. To confirm the transaction, the transaction initiator service calls the transaction coordinator, which then calls each participant service.

To confirm a TCC transaction:

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Transactions** tab.
3. In the **Transaction Type** drop-down list, select **TCC** to view the transaction details for applications using the TCC transaction protocol.
4. Click **Action** , and then click **Confirm**.

8.9.2 Cancel a TCC Transaction

Cancels the resources that are held in a reserved state in the TCC transaction. To cancel the transaction, the transaction initiator service calls the transaction coordinator, which then calls each participant service.

To cancel an TCC transaction:

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Transactions** tab.
3. In the **Transaction Type** drop-down list, select **TCC** to view the transaction details for applications using the TCC transaction protocol.
4. Click **Action** , and then click **Cancel**.

8.10 Manage Saga Transactions

- [Complete a Saga Transaction](#)
To complete the transaction, the transaction initiator service calls Transaction Manager for Microservices, and then the Transaction Manager for Microservices calls each participant service's complete callback URI.
- [Compensate a Saga Transaction](#)
Compensates or cancels a Saga transaction. To compensate the transaction, the transaction initiator service calls the transaction coordinator, and then the transaction coordinator calls each participant service's compensate callback URI.

8.10.1 Complete a Saga Transaction

To complete the transaction, the transaction initiator service calls Transaction Manager for Microservices, and then the Transaction Manager for Microservices calls each participant service's complete callback URI.

To complete a Saga transaction:

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Transactions** tab.
3. In the **Transaction Type** drop-down list, select **Saga** to view the transaction details for applications using the **Saga** transaction protocol.
4. Click **Action** , and then click **Complete**.

8.10.2 Compensate a Saga Transaction

Compensates or cancels a Saga transaction. To compensate the transaction, the transaction initiator service calls the transaction coordinator, and then the transaction coordinator calls each participant service's compensate callback URI.

To compensate a Saga transaction:

1. Sign in to the MicroTx Distributed Transactions console. See [Access the Web Console](#).
2. Click the **Transactions** tab.
3. In the **Transaction Type** drop-down list, select **Saga** to view the transaction details for applications using the **Saga** transaction protocol.
4. Click **Action** , and then click **Compensate**.

Deploy Sample Applications

Using samples is the fastest way for you to get familiar with MicroTx. This is an optional task. After installing MicroTx Distributed Transactions, you may want to run sample applications and then use distributed tracing to understand how requests flow between MicroTx Distributed Transactions and the microservices.

Sample applications are microservices that demonstrate how you can develop your services for participating in different transaction protocols using MicroTx Distributed Transactions. The code of the sample applications includes the MicroTx Distributed Transactions libraries. You can use the sample applications as a reference while integrating the MicroTx Distributed Transactions libraries with your application.

You can use one of the following options to run the sample applications.

- Use the `runme.sh` script file to install Transaction Manager for Microservices (MicroTx) in a runtime environment, and then quickly run sample applications. See About the `runme.sh` Script in *Transaction Manager for Microservices Quick Start Guide*.
- Clone the sample application code from the git repository, <https://github.com/oracle-samples/microtx-samples>.
This repository contains the source code for sample applications of different transaction protocols, XA, Saga, and TCC, in individual folders. The source code of the sample applications is already integrated with the MicroTx libraries. Since a sample application consists of multiple microservices, each folder contains the source code for all the sample microservices and files required by Helm to install the microservices. For details to set up and run the sample applications, refer to the readme files located in the respective folders.
- Use the step-by-step instructions provided in the hands-on labs to set up and run the sample applications. See [MicroTx LiveLabs](#).

Topics

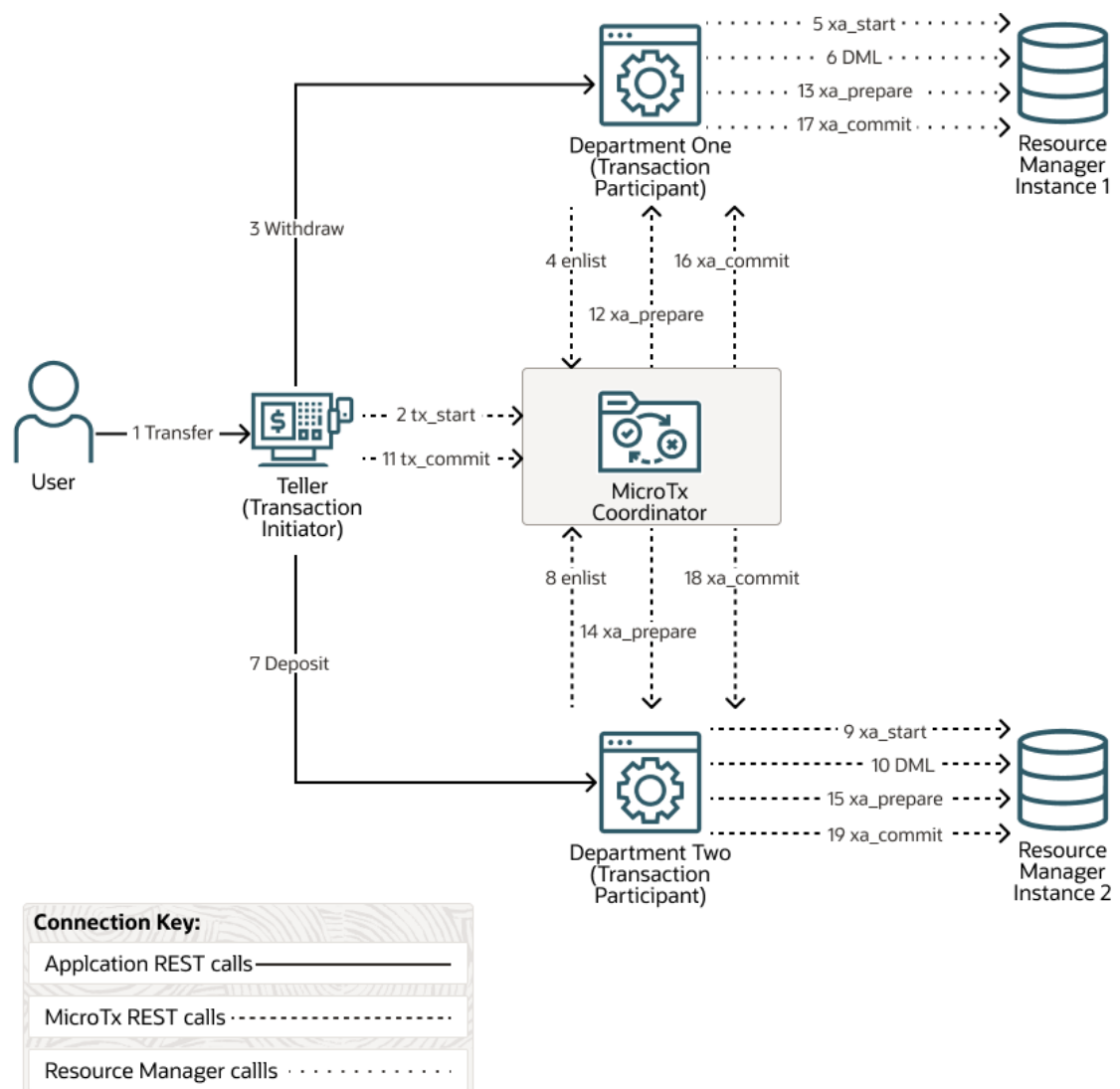
- [About XA Sample Application](#)
The sample XA application implements a scenario where a Teller application initiates an amount transfer from one department to another by creating an XA transaction. The two departments in the organization are Department One (Dept 1) and Department Two (Dept 2).
- [About the Sample Saga Application](#)
Use the sample Saga application to book a trip, which consists of booking a hotel room and a flight.
- [About the Sample TCC Application](#)
Let's use the sample TCC application that's available in the installation bundle to understand how microservices and MicroTx Distributed Transactions interact with each other in a TCC transaction.

9.1 About XA Sample Application

The sample XA application implements a scenario where a Teller application initiates an amount transfer from one department to another by creating an XA transaction. The two departments in the organization are Department One (Dept 1) and Department Two (Dept 2).

MicroTx Distributed Transactions implements the XA transaction. Within the XA transaction, all actions such as withdraw and deposit either succeed, or they all are rolled back in case of a failure of any one or more actions.

The following image shows a sample XA application deployment which consists of polyglot participant microservices.



- MicroTx Distributed Transactions coordinator manages transactions amongst the participant services.
- Teller microservice initiates the transactions, so it is called an XA transaction initiator service. The user interacts with this microservice to transfer money between departments One and Two. When a new request is created, the helper method that is exposed in the MicroTx Distributed Transactions library runs the `begin()` method for XA transaction to

start the XA transaction at the Teller microservice. This microservice also contains the business logic to issue the XA commit and roll back calls.

- Department One and Department Two participate in the transactions, so they are called as XA participant services. The MicroTx Distributed Transactions library includes headers that enable the participant services to automatically enlist in the transaction. These microservices expose REST APIs to get the account balance and to withdraw or deposit money from a specified account. They also use resources from resource manager.

Resource managers manage stateful resources such as databases, queuing or messaging systems, and caches.

The service must meet ACID requirements, so an XA transaction is initiated and both withdraw and deposit are called in the context of this transaction.

The next topic describes how the microservices and MicroTx Distributed Transactions communicate during an XA transaction.

- [Scenario: Withdraw and Deposit an Amount](#)
The following steps describe an example sequence for the successful path of an XA transaction when you run the sample application. Let us consider a scenario, where a user places a request to withdraw an amount from Department One and deposit that amount into Department Two. In case of failures, the initiating application calls a rollback instead of a commit.

9.1.1 Scenario: Withdraw and Deposit an Amount

The following steps describe an example sequence for the successful path of an XA transaction when you run the sample application. Let us consider a scenario, where a user places a request to withdraw an amount from Department One and deposit that amount into Department Two. In case of failures, the initiating application calls a rollback instead of a commit.

It is assumed that Department One and Department Two use XA-compliant resource managers.

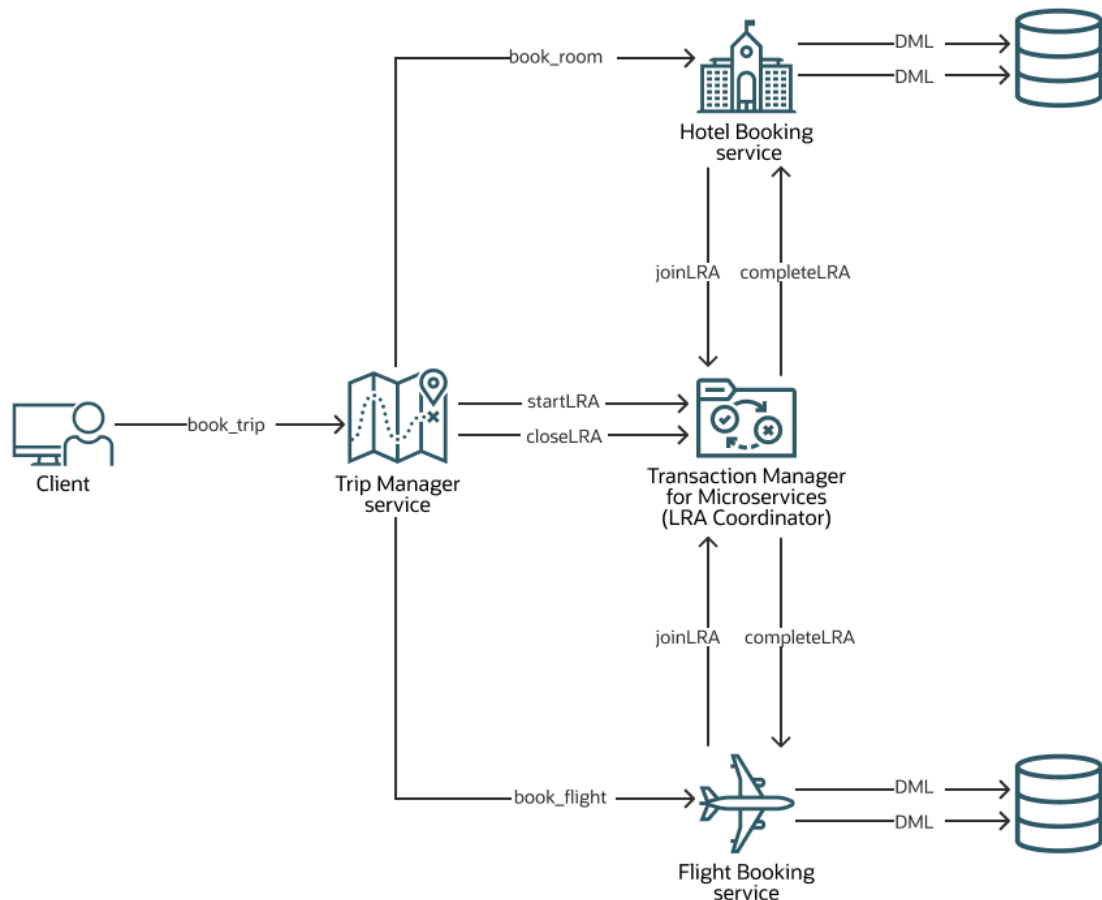
1. User places a request to transfer an amount from Department One to Department Two.
2. The Teller service initiates the transaction, when a user places a request to withdraw an amount from Department One. The transaction initiator service, Teller, makes a call to MicroTx Distributed Transactions to begin an XA transaction. MicroTx Distributed Transactions creates a new global transaction ID (GTRID) to track the transaction, writes the GTRID to the data store, and returns the GTRID to the transaction initiator service.
3. The Teller sends a request to Department One to withdraw an amount.
4. The transaction participant service, Department One enlists to MicroTx Distributed Transactions with the same GTRID. MicroTx Distributed Transactions may have to interact with multiple participant services to successfully complete a transaction. MicroTx Distributed Transactions also creates a branch ID that is unique to each participant service. The XID contains both the GTRID and branch ID, so the XID is unique for each participant service.
5. In XA protocol, MicroTx Distributed Transactions manages the communication between participant microservices, such as Department One, and the resource manager. Department One must use the MicroTx client libraries which registers callbacks and provides implementation of the callbacks for the resource manager.
6. Department One performs the DML operation to withdraw the amount, and then returns a response.

7. The Teller service initiates another request to deposit an amount to Department Two.
8. The transaction participant service, Department Two enlists to MicroTx Distributed Transactions with the same GTRID. MicroTx Distributed Transactions coordinator creates a branch ID that is unique to Department Two.
9. Department Two communicates with the resource manager using the integrated MicroTx Distributed Transactions library.
10. Department Two performs the DML operation to deposit the amount, and then returns a response.
11. The Teller service commits the transaction only if both the requests, that is, the request to Department One and the request to Department Two, are executed successfully. In case of any failure, the Teller service calls rollback instead of commit. Teller tracks the commit transaction using the same GTRID that was used by Department One and Two.
12. MicroTx Distributed Transactions coordinator prepares the participant service, Department One, to commit the transaction.
13. MicroTx Distributed Transactions coordinator calls Department One. The participant microservice in turn uses the integrated MicroTx Distributed Transactions library to send a request to prepare the resource manager.
14. MicroTx Distributed Transactions coordinator prepares the participant service, Department Two, to commit the transaction.
15. Coordinator sends a request to Department Two. The participant microservice in turn uses the integrated MicroTx Distributed Transactions library to send a request to prepare the resource manager.
16. The coordinator sends a commit request to the participant services after the prepare phase is completed successfully. The participant services in turn send a request to the resource manager using the integrated MicroTx Distributed Transactions library. The coordinator returns a response to the Teller service which completes the transaction.

9.2 About the Sample Saga Application

Use the sample Saga application to book a trip, which consists of booking a hotel room and a flight.

The following figure shows a sample Saga application, which contains several microservices, to demonstrate how you can use MicroTx Distributed Transactions to manage Saga transactions. Each microservice in the sample application performs a different task. One microservice books a trip, another books a flight, and a third microservice books a hotel. MicroTx Distributed Transactions coordinates the transactions between these microservices.



The sample Saga application consists of the following polyglot microservices:

- **MicroTx Distributed Transactions (Saga Coordinator)** coordinates the transaction between the sample microservices.
- **Trip Manager service** is the transaction initiator service, where the Saga transaction starts. While booking a trip, this service calls the flight and hotel services for booking a flight and hotel respectively. The Trip Manager exposes the APIs to book both the hotel and flight and to cancel the booking. Either both hotel and flight are booked successfully or both bookings are canceled if there is a failure.
- **Hotel Booking service** exposes APIs to book a hotel room and also to cancel the booking in case of any failure. It is called by the Trip Manager service to reserve a room. As it is called within the context of an existing Saga, it enlists itself and provides callback URIs that the Saga coordinator uses to complete or compensate the room reservation.
- **Flight Booking service** exposes APIs to book a flight ticket and also to cancel the booking in case of any failure. It is called by the Trip Manager service to book a flight ticket. As it is called within the context of an existing Saga, it enlists itself and provides callback URIs that the Saga coordinator uses to complete or compensate the flight reservation.
- **Trip client** is the user interface which you can use to confirm or cancel the booking. It does not participate in the Saga transaction. It is provided as sample client service which calls microservices to perform a distributed transaction that uses the Saga protocol.

The source code for the Saga sample application is available in the `lra` folder in the [microtx-samples](#) GitHub repository. MicroTx Distributed Transactions libraries are included in the code of the sample application microservices. The services communicate with each other through the exposed REST endpoints while using the MicroTx Distributed Transactions libraries.

When you run the application, it makes a provisional booking by reserving a hotel room and flight ticket. Only when you provide approval to confirm the booking, the booking of the hotel room and flight ticket is confirmed. If you cancel the provisional booking, the hotel room and flight ticket that was blocked is released and the booking is canceled. By default, the flight service permits only two confirmed bookings. To enable you to test the failure scenario, the flight service sample application rejects any additional booking requests that are made after two confirmed bookings. This leads to the cancellation (compensation) of a provisionally booked hotel within the trip and the trip is not booked.

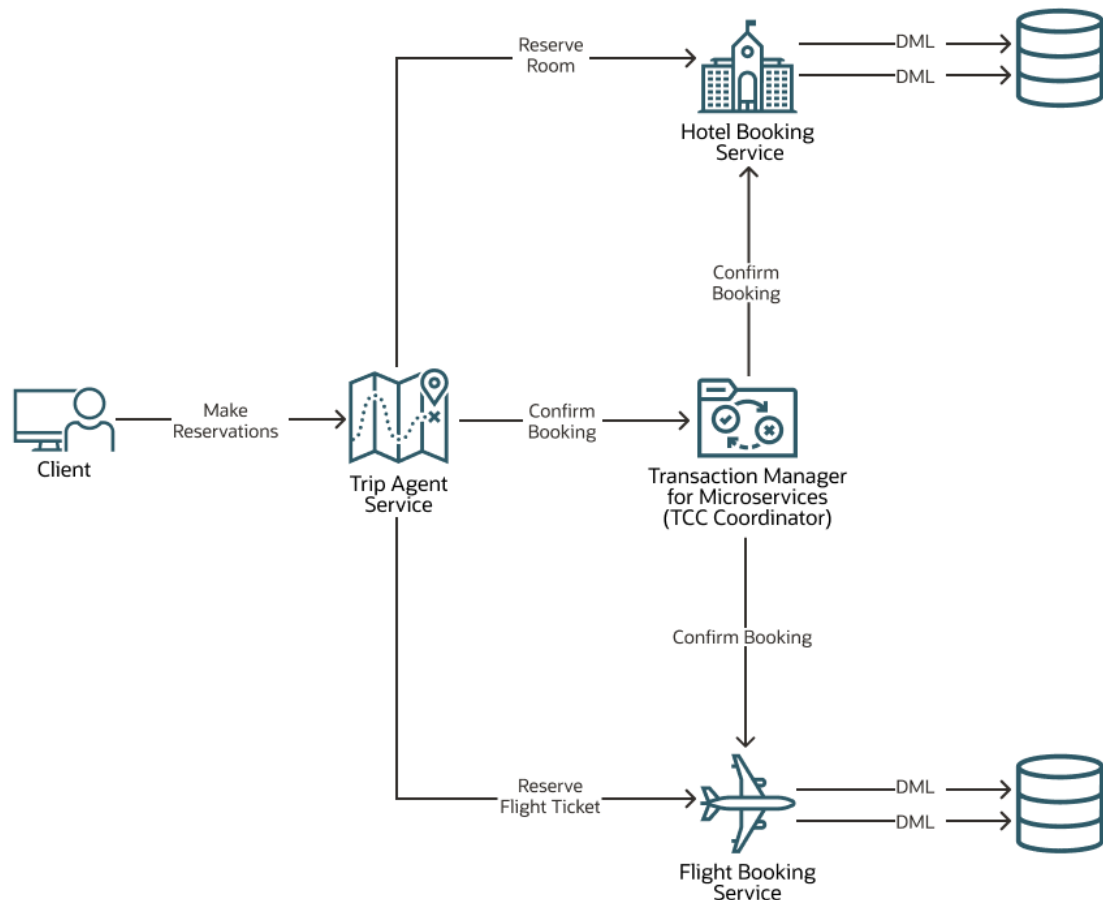
9.3 About the Sample TCC Application

Let's use the sample TCC application that's available in the installation bundle to understand how microservices and MicroTx Distributed Transactions interact with each other in a TCC transaction.

The sample application uses the TCC transaction protocol and MicroTx Distributed Transactions to coordinate the transactions. The MicroTx Distributed Transactions libraries are already integrated with the sample application code.

The sample TCC application implements a scenario where the travel agent microservice books a trip, flight booking service books a flight, and the hotel booking microservice books a hotel. The travel agent service accesses both the flight and hotel booking services. When a customer books a flight and a hotel, the booking is reserved until either the customer completes the payment and confirms the booking. In case of any failure, the reserved resources are canceled and the resources are returned back to the inventory.

The following figure shows a sample TCC application, which contains several microservices, to demonstrate how you can use MicroTx Distributed Transactions to manage TCC transactions.



The sample TCC application consists of the following microservices:

- MicroTx Distributed Transactions (TCC Coordinator)
- Travel Agent service is the transaction initiator service, where the TCC transaction starts. It provides APIs to book and cancel a hotel room and a flight ticket. While booking a trip, this service calls the flight booking and hotel booking services. It also sends the confirm or cancel call to the transaction participant services to finalize the transaction.
- Hotel Booking service participates in the transactions, so it is also called a transaction participant service. It provides APIs to confirm and cancel a hotel room booking.
- Flight Booking service participates in the transactions, so it is also called a transaction participant service. It provides APIs to confirm and cancel a flight ticket booking.

The code for the TCC sample application is available in the `tcc` folder in the [microtx-samples](#) GitHub repository. The sample TCC application code is available in Node.js and Java. When you run the sample application, build all the three sample microservices of either Node.js or Java. Don't try to run the Travel Agent service in Java with Hotel Booking service in Node.js.

Monitor and Trace Transactions

Monitor transactions to understand how the requests flow between MicroTx Distributed Transactions and the microservices and how MicroTx Distributed Transactions manages the transaction.

Prometheus sends a request to each Transaction Manager for Microservices pod to collect metrics from each pod. Use Grafana to visualize the metrics data collected into Prometheus. View the logs emitted by Transaction Manager for Microservices to troubleshoot any issues that may arise.

- [Trace](#)
Use distributed tracing to understand how requests flow between MicroTx Distributed Transactions and the microservices. Use tools, such as Kiali and Jaeger, to track and trace distributed transactions in MicroTx Distributed Transactions.
- [Monitor Performance](#)
Use Prometheus and Grafana to view performance metrics for your services. You can use these metrics to monitor your transactions and the health of the MicroTx Distributed Transactions coordinator.
- [Install and Configure Alertmanager](#)
Prometheus scrapes data from the MicroTx coordinator at regular intervals. Prometheus Alertmanager sends notifications based on the specified thresholds and rules. If you have configured an email ID to receive notifications, Alertmanager sends notifications to the configured email ID.
- [Logs](#)
The MicroTx Distributed Transactions coordinator and the MicroTx Distributed Transactions client libraries for Java, Spring Boot, Node.js, and Python generate logs.

10.1 Trace

Use distributed tracing to understand how requests flow between MicroTx Distributed Transactions and the microservices. Use tools, such as Kiali and Jaeger, to track and trace distributed transactions in MicroTx Distributed Transactions.

Istio is a service mesh that provides a separate infrastructure layer to handle inter-service communication. Network communication is abstracted from the services themselves and is handled by proxies. Istio uses a sidecar design, which means that the communication proxies run in their own containers beside every service container. Envoy is the proxy that is deployed as a sidecar inside the microservices container. All communication inside the service mesh is done through the Envoy proxies. The Envoy proxies automatically generate trace spans on behalf of the microservices they proxy, requiring only that the services forward the appropriate request context. See <https://istio.io/latest/docs/concepts/observability/>. Istio supports many tracing backends, such as Zipkin, Jaeger, Lightstep, and Datadog.

Note

The steps provided in this section are specific to an environment where MicroTx Distributed Transactions and Istio are deployed in a Kubernetes cluster. Use the instructions provided in this section only for test or development environments. These instructions are not meant for production environments.

For more information, refer to the Kiali and Jaeger documentation.

- [Install Jaeger](#)
When you download the Istio installation bundle, it contains `jaeger.yaml`, a basic sample installation to quickly get Jaeger up and running. The `jaeger.yaml` file is available in the `samples/addons` folder at the location where you have downloaded the Istio installation files.
- [Perform Distributed Tracing with Jaeger](#)
- [Install Kiali](#)
When you download the Istio installation bundle, it contains `kiali.yaml`, a basic sample installation to quickly get Kiali up and running. The `kiali.yaml` file is available in the `samples/addon` folder at the location where you have downloaded the Istio installation files.
- [List of Trace Headers](#)
When you want to trace the transaction from end-to-end, set `oracle.tmm.PropagateTraceHeaders` to `true`. This propagates the trace headers for all incoming and outgoing requests.

10.1.1 Install Jaeger

When you download the Istio installation bundle, it contains `jaeger.yaml`, a basic sample installation to quickly get Jaeger up and running. The `jaeger.yaml` file is available in the `samples/addons` folder at the location where you have downloaded the Istio installation files.

Alternatively, install Jaeger separately. See <https://istio.io/latest/docs/ops/integrations/jaeger/>.

To install Jaeger using the YAML file that is available in the Istio package directory:

1. Move to the Istio package directory. For example, if the package is `istio-1.15.0`:

```
cd istio-1.15.0
```

2. Install Jaeger.

```
kubectl apply -f samples/addons/jaeger.yaml
```

Sample response

```
deployment.apps/jaeger created
service/tracing created
service/zipkin created
service/jaeger-collector created
```

3. Run the following command to verify that Jaeger was installed.

```
kubectl get all -n istio-system
```

Sample response

NAME	READY	STATUS	RESTARTS
AGE			
pod/istio-ingressgateway-6cc856bd7d-qcwk7	1/1	Running	0
10d			
pod/istiod-945b9f699-frff5	1/1	Running	0
10d			
pod/jaeger-c4fdf6674-wqxb	1/1	Running	0
11m			

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
PORT(S)		AGE	
service/istio-ingressgateway	LoadBalancer	10.97...	<pending>
15021:30651/TCP,80:31635/TCP,443:32196/TCP		10d	
service/istiod	ClusterIP	10.100...	<none>
15010/TCP,15012/TCP,443/TCP,15014/TCP		10d	
service/jaeger-collector	ClusterIP	10.110...	<none>
14268/TCP,14250/TCP,9411/TCP		11m	
service/tracing	ClusterIP	10.107...	<none>
80/TCP,16685/TCP		11m	
service/zipkin	ClusterIP	10.106...	<none>
9411/TCP		11m	

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/istio-ingressgateway	1/1	1	1	10d
deployment.apps/istiod	1/1	1	1	10d
deployment.apps/jaeger	1/1	1	1	11m

NAME	DESIRED	CURRENT
READY AGE		
replicaset.apps/istio-ingressgateway-6cc856bd7d	1	1
1 10d		
replicaset.apps/istiod-945b9f699	1	1
1 10d		
replicaset.apps/jaeger-c4fdf6674	1	1
1 11m		

NAME	TARGETS	MINPODS	MAXPODS
REFERENCE REPLICAS AGE			
horizontalpodautoscaler.autoscaling/istio-ingressgateway			Deployment/
istio-ingressgateway <unknown>/80%	1	5	1 10d
horizontalpodautoscaler.autoscaling/istiod			Deployment/
istiod <unknown>/80%	1	5	1 10d

10.1.2 Perform Distributed Tracing with Jaeger

To understand how to perform distributed tracing using Jaeger, let us consider the sample application for XA.

The sample application implements a scenario where an Accounts department application transfers money from one department to another by creating an XA transaction. The two

departments in the organization are Dept 1 and Dept 2. For more details about the sample XA application that is available in the installation bundle, see [XA Transaction Protocol](#).

Before you perform distributed tracing, ensure that you have deployed the application and initiated a transaction.

1. Open the Jaeger UI using `istioctl`.

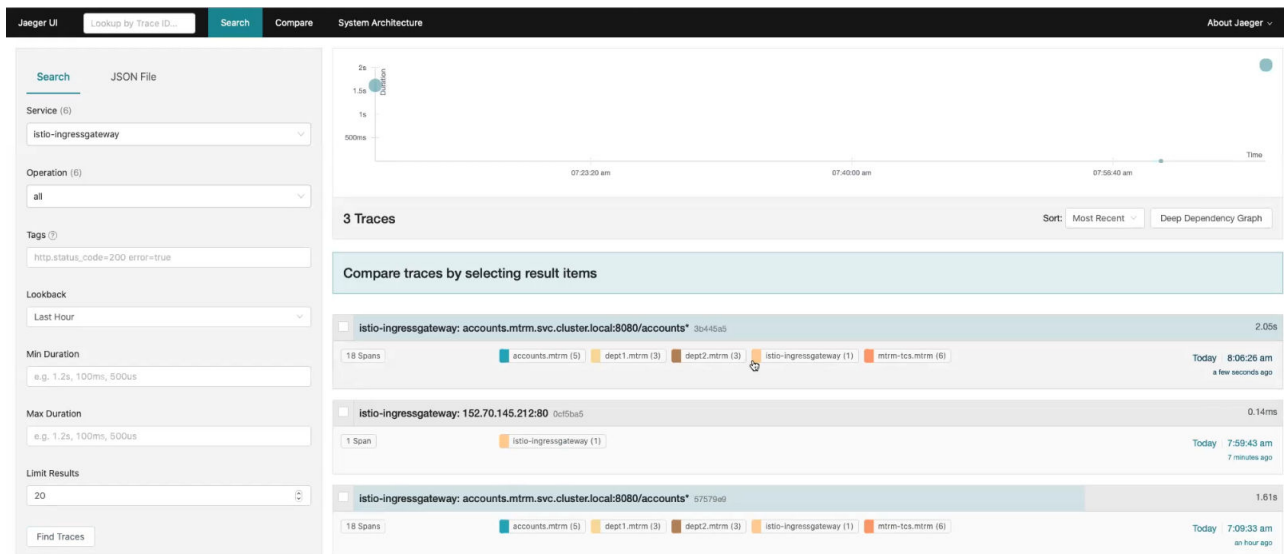
```
istioctl dashboard jaeger
```

2. Perform a transaction using your application.

In case of the sample XA application, use the Accounts service to withdraw an amount from Dept 1 and deposit that amount to Dept 2.

3. In the Jaeger UI, click the **Search** tab.
4. In the **Service** drop-down list, select **istio-ingressgateway**.
5. Click **Find Traces**, and then locate the trace that is time-stamped as **a few seconds ago**.

The trace for your latest transaction using the sample XA application is displayed as shown in the following figure.



6. Click the trace to view more details.
7. Under **Service & Operation**, view the flow of all the requests.

The Istio ingress gateway receives the request and forwards it to the Accounts service, which is the initiator service. From Accounts service, a call was sent to TCS to begin the transaction.

10.1.3 Install Kiali

When you download the Istio installation bundle, it contains `kiali.yaml`, a basic sample installation to quickly get Kiali up and running. The `kiali.yaml` file is available in the `samples/addon` folder at the location where you have downloaded the Istio installation files.

Alternatively, install Kiali separately. See <https://kiali.io/docs/installation/>.

To install Kiali using the YAML file that is available in the Istio package directory:

1. Move to the Istio package directory. For example, if the package is istio-1.15.0:

```
cd istio-1.15.0
```

2. Install Kiali.

```
kubectl apply -f samples/addons/kiali.yaml
```

Sample response

```
serviceaccount/kiali created
configmap/kiali created
clusterrole.rbac.authorization.k8s.io/kiali-viewer created
clusterrole.rbac.authorization.k8s.io/kiali created
clusterrolebinding.rbac.authorization.k8s.io/kiali created
role.rbac.authorization.k8s.io/kiali-controlplane created
rolebinding.rbac.authorization.k8s.io/kiali-controlplane created
service/kiali created
deployment.apps/kiali created
```

3. Run the following command to verify that Kiali was installed.

```
kubectl -n istio-system get svc kiali
```

Sample response

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kiali	ClusterIP	10.100.214.26	<none>	20001/TCP,9090/TCP	62s

4. Open the Kiali dashboard.

```
istioctl dashboard kiali
```

10.1.4 List of Trace Headers

When you want to trace the transaction from end-to-end, set `oracle.tmm.PropagateTraceHeaders` to true. This propagates the trace headers for all incoming and outgoing requests.

The following table lists a few of the trace headers that are propagated.

Name of the headers	Description
x-request-id	All applications must propagate this header. This header is included in access log statements and it is used for consistent trace sampling and log sampling decisions in Istio.
oracle-tmm-tx-token authorization refresh-token oracle-tmm-authz-token	These MicroTx Distributed Transactions-specific headers must be propagated for running the MicroTx Distributed Transactions API calls from the library.

Name of the headers	Description
end-user	This header is specific to the application and you can forward this header.
x-ot-span-context	Propagate this header if you are using Lightstep tracing in Istio. See https://istio.io/latest/docs/tasks/observability/distributed-tracing/lightstep/ .
x-datadog-trace-id x-datadog-parent-id x-datadog-sampling-priority	Propagate these headers if you are using Datadog tracing.
traceparent tracestate	These are W3C trace context headers. They are compatible with OpenCensus Agent and Stackdriver configurations for Istio.
x-cloud-trace-context	This is a Cloud Trace context header. It is compatible with OpenCensus Agent and Stackdriver configurations for Istio.
grpc-trace-bin	This is a gRPC binary trace context header. It is compatible with OpenCensus Agent and Stackdriver configurations for Istio.
x-b3-traceid x-b3-spanid x-b3-parentspanid x-b3-sampled x-b3-flags	These are B3 trace context headers. They are compatible with Zipkin, OpenCensus Agent, and Stackdriver configurations for Istio.

10.2 Monitor Performance

Use Prometheus and Grafana to view performance metrics for your services. You can use these metrics to monitor your transactions and the health of the MicroTx Distributed Transactions coordinator.

The MicroTx Distributed Transactions coordinator exposes metrics in a format that can be easily read by Prometheus. Prometheus sends a request to each MicroTx Distributed Transactions pod to collect metrics from each pod in the Kubernetes cluster. Prometheus

monitors and gathers metrics about the MicroTx Distributed Transactions coordinator and records metrics that tracks the health of all applications within the Istio service mesh.

Use Grafana to visualize the metrics data collected into Prometheus. You can import MicroTx Distributed Transactions dashboards to view metrics.

- [Set Up Prometheus and Grafana](#)
Set up Prometheus and Grafana in the Kubernetes cluster where you have installed MicroTx Distributed Transactions to monitor MicroTx Distributed Transactions.
- [View Metrics with Prometheus and Grafana](#)
View MicroTx Distributed Transactions coordinator metrics with Prometheus and Grafana.

10.2.1 Set Up Prometheus and Grafana

Set up Prometheus and Grafana in the Kubernetes cluster where you have installed MicroTx Distributed Transactions to monitor MicroTx Distributed Transactions.

Before you begin, ensure that you have installed Istio and MicroTx Distributed Transactions.

1. Set the `metrics` property to `true` in the `values.yaml` file, Helm Chart used to deploy the MicroTx Distributed Transactions coordinator. You must set this property to enable Prometheus to scrape the metrics logs of the MicroTx Distributed Transactions coordinator.

The following code snippet provides a sample value for the `metrics` property in the `values.yaml` file.

```
metrics:
  enabled: true
```

2. Open the `prometheus.yaml` file in any code editor.

When you download the Istio installation bundle, it contains `prometheus.yaml`, a basic sample installation to quickly get Prometheus up and running. The `prometheus.yaml` file is available in the `samples/addon` folder at the location where you have downloaded the Istio installation files.

3. Under `scrape_configs`, add the following lines of code and save the changes.

```
scrape_configs:
- job_name: tcs-metric
  static_configs:
  - targets:
    - 192.0.2.1
```

Where, `192.0.2.1` is the external IP address of the Istio ingress gateway. For details about how to note down this value, see [Find IP Address of Istio Ingress Gateway](#).

4. Run the following command to install Prometheus.

```
kubectl apply -f samples/addons/prometheus.yaml
```

5. Run the following command to verify that Prometheus was installed.

```
kubectl -n istio-system get svc prometheus
```

6. Ensure that Prometheus service discovery is able to discover the MicroTx Distributed Transactions host. Refer to the Prometheus documentation for more details.

To ensure that Prometheus can discover the MicroTx Distributed Transactions host, check the Targets in Prometheus or search for any one of the MicroTx Distributed Transactions metrics in the Prometheus UI.

7. Install Grafana.

```
kubectl apply -f samples/addons/grafana.yaml
```

When you download the Istio installation bundle, it contains `grafana.yaml`, a basic sample installation to quickly get Grafana up and running. The `grafana.yaml` file is available in the `samples/addon` folder at the location where you have downloaded the Istio installation files. Alternatively, install Grafana separately. See <https://grafana.com/docs/grafana/latest/installation/kubernetes/>.

8. Run the following command to verify that Grafana was installed.

```
kubectl -n istio-system get svc grafana
```

9. Start the Grafana dashboard.

```
istioctl dashboard grafana
```

The Grafana dashboard opens in a new browser. Note down the link to access the Grafana dashboard.

10. Create a Prometheus data source in Grafana, so that the MicroTx Distributed Transactions metrics scraped by Prometheus are displayed in the Grafana dashboard.

11. Import MicroTx Distributed Transactions dashboard by importing the JSON file which contains metrics and details about MicroTx Distributed Transactions. This file is located at `<installation_directory>/otmm-<version>/otmm/dashboards/microtx-dashboard.json`.

For information about importing dashboards in Grafana, see <https://grafana.com/docs/grafana/latest/dashboards/export-import/#import-dashboard>.

10.2.2 View Metrics with Prometheus and Grafana

View MicroTx Distributed Transactions coordinator metrics with Prometheus and Grafana.

Before you begin, ensure that you have installed and configured Prometheus and Grafana.

1. In any browser, enter the following URL to view metrics with Prometheus.

Syntax of URL

```
https://externalHostname/metrics
```

Where, `externalHostname`: The IP address of the load balancer of the Istio ingress gateway. See [Find IP Address of Istio Ingress Gateway](#). For example, `192.0.2.1`.

2. View the metrics of the MicroTx Distributed Transactions coordinator.

Prometheus provides value for different metrics and a description for each metric.

3. Visualize the metrics scraped by Prometheus in the MicroTx Distributed Transactions dashboard in Grafana.

10.3 Install and Configure Alertmanager

Prometheus scrapes data from the MicroTx coordinator at regular intervals. Prometheus Alertmanager sends notifications based on the specified thresholds and rules. If you have configured an email ID to receive notifications, Alertmanager sends notifications to the configured email ID.

Alerts are sent based on the thresholds or rules mentioned in the `prometheus.yaml` file. For example, users are notified when the MicroTx Distributed Transactions service or database service is not available. You can customize the alerts. To add a custom alert, see <https://prometheus.io/docs/alerting/latest/notifications/>.

1. Install Prometheus. See [Set Up Prometheus and Grafana](#).
2. Open the `prometheus.yaml` file.

In Kubernetes environments, use the `prometheus.yaml` which is located at `installation_directory/MicroTx-RELEASE/MicroTx/visualization/alert/kubernetes`. The MicroTx coordinator uses the configuration details provided in the `prometheus.yaml` to monitor and manage alerts. This file also defines the rules to generate alerts.

This file contains details about the instance from which Prometheus scrapes the metrics and it defines the threshold for which alert messages are sent. For description about each property and the permitted values, see <https://prometheus.io/docs/prometheus/latest/configuration/configuration/>.

To install Prometheus in your local environment, use the `prometheus.yaml` and `rules.yaml` files which are located at `installation_directory/MicroTx-RELEASE/MicroTx/visualization/alert/local`. The MicroTx coordinator uses the configuration details provided in the `prometheus.yaml` to monitor and manage alerts. The `rules.yaml` file defines the rules to generate alerts.

Provide values for the following properties.

- Enter the `scrape_interval` and `evaluation_interval` to specify the time interval at which Prometheus scrapes the metrics from the coordinator.

```
server:
  global:
    scrape_interval: 1m
    evaluation_interval: 1m
```

- Enter a unique value for `job_name` and specify the instance that you want to connect to. Prometheus scrapes the metrics logs from the specified instance.

```
scrape_configs:
- job_name: prometheus
  static_configs:
  - targets:
    - localhost:9090
- job_name: "tcs"
  static_configs:
  - targets:
    - otmm-tcs.otmm.svc.cluster.local:9000
- job_name: kube-state-metrics
  metrics_path: /metrics
  static_configs:
```

- targets:
 - prometheus-kube-state-metrics.otmm.svc.cluster.local:8080

Where,

- scrape_config is the name of the job in Prometheus.
- otmm-tcs.otmm.svc.cluster.local:9000 is the name of the MicroTx Distributed Transactions coordinator service and the port number at which the service is running. Alert messages are generated for this service.
- prometheus-kube-state-metrics.otmm.svc.cluster.local:8080 is the target from which you want to scrape the metrics. Prometheus exposes a set of metrics by default. When you install Prometheus in a cluster, a set of services are deployed which includes prometheus-kube-state-metrics. Use this service to query the state, health, or performance of any Kubernetes resource.

3. Run the following commands to download the Helm Charts from the prometheus-community repo.

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
```

4. Reinstall Prometheus using the prometheus.yaml file that you have downloaded.

```
helm install -f prometheus.yaml prometheus prometheus-community/prometheus
```

5. Download the Prometheus Alertmanager from <https://prometheus.io/download/#alertmanager>.

6. Run the following command to install Prometheus Alertmanager.

```
helm install [RELEASE_NAME] prometheus-community/alertmanager
```

Where, [RELEASE_NAME] is the release number of Prometheus Alertmanager that you want to install.

You can configure to receive alerts on Slack or email. See the Prometheus Alertmanager documentation.

If you have installed Prometheus and Alertmanager in your local environment, you can access Prometheus at <http://localhost:9090/> and Alertmanager at <http://localhost:9093/>.

10.4 Logs

The MicroTx Distributed Transactions coordinator and the MicroTx Distributed Transactions client libraries for Java, Spring Boot, Node.js, and Python generate logs.

To view the logs emitted by MicroTx Distributed Transactions in a Kubernetes cluster, you can set up the Elasticsearch, Fluentd or Fluent bit, and Kibana (EFK), Elasticsearch, LogStash, and Kibana (ELK) stack, or any other application to aggregate and view the log files of MicroTx Distributed Transactions coordinator and applications at a single place.

- [Define the Syntax of Log Messages](#)
Specify the details that you want to record in the logs and the syntax of the log messages.

- [Sample Log Messages](#)
View example log messages and their syntax.
- [Enable Logs for MicroTx Distributed Transactions Node.js Library](#)
By default, logging is enabled for the MicroTx Distributed Transactions Coordinator and Java applications that use the MicroTx Distributed Transactions client library. However, for Node.js applications that use the MicroTx Distributed Transactions client libraries, you must enable logging.

10.4.1 Define the Syntax of Log Messages

Specify the details that you want to record in the logs and the syntax of the log messages.

To view the logs emitted by MicroTx Distributed Transactions in a Kubernetes cluster, you can set up one of the following stacks or any other application to aggregate and view the log files of MicroTx Distributed Transactions coordinator and applications at a single place.

- Elasticsearch, Fluentd or Fluent bit, and Kibana (EFK) stack
- Elasticsearch, LogStash, and Kibana (ELK) stack

For information about setting up EFK or ELK stack, refer to the application's product documentation.

After setting up the EFK or ELK stack, you must perform the following steps once to specify the details that you want to record in the logs and the syntax of the log messages.

1. Define the parser filters for the log aggregator. The following example defines the parsers for Fluentbit. The Regex field provides the syntax of the log message and the details that are logged. Similarly, you can define the parsers for any log aggregator that you want to use. Refer to the log aggregator's product documentation for more details.

```
[PARSER]
  Name mtxnnodejslib
  Format regex
  Regex ^(?<timestamp>\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}\.\d{3}Z)
\s+::\s+(?<transactionId>[a-f0-9-]+\s+\s+(?<message>.+)$
  Time_Key timestamp
  Time_Format %Y-%m-%dT%H:%M:%S.%zZ

[PARSER]
  Name mtxlrasingbootlib
  Format regex
  Regex ^(?<timestamp>\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\.\d{3})\s+
(?<log_level>\w+)\s+(?<numeric_value>\d+)\s+---\s+\[(?<thread_info>[\w-]+)
\]\s+(?<logger_name>\S+)\s+:\s+(?<transactionId>\S+)\s+\s+(?<message>.+)$
  Time_Key timestamp
  Time_Format %Y-%m-%d %H:%M:%S.%z

[PARSER]
  Name mtxhelidon
  Format regex
  Regex ^(?<timestamp>\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\.\d{3})\s+
(?<log_level>\w+)\s+---\s+\[(?<thread_name>[\^]]+)\s+\s+(?<logger_name>[\^:]
+)\s+:\s+(?<transactionId>[a-f0-9-]+\s+\s+(?<log_message>.+)$
  Time_Key timestamp
  Time_Format %Y-%m-%d %H:%M:%S.%z

[PARSER]
```

```

        Name mtxxaspringbootlib
        Format regex
        Regex ^(?<timestamp>\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\.\d{3})\s+
        (?<log_level>\w+)\s+\d+\s+---\s+\[(?<thread_name>[^\]]+)\]\s+(?
        <logger_name>[^\:]+\s+:\s+(?<transactionId>[a-f0-9-]+\s+:\s+(?
        <log_message>.+)$
        Time_Key timestamp
        Time_Format %Y-%m-%d %H:%M:%S.%Z

```

To view the example logs that correspond to the syntax you have specified in the parser, see [Sample Log Messages](#).

2. For the MicroTx Distributed Transactions Java library for Helidon applications, create an XML file, such as `log4j2.xml`, in your application's resources folder to specify the syntax of the log messages. For example, `src/main/resources/log4j2.xml`.
 - a. The following code sample demonstrates how you can specify the position of the various fields in the log message.

```

<Console name="MicroTxLib" target="SYSTEM_OUT">
    <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS}
    %-5level --- [%t] %-50.70C : [microtx-log] %X{microtx-txnId} :
    %msg%n%throwable"/>
</Console>

```

Where,

- *MicroTxLib* Note down this name as you will provide this name later in the Logger section to define the packages that you want to log.
 - *PatternLayout* specifies the syntax of the log messages.
 - `[microtx-log]` is prefixed to the log messages generated by the MicroTx Distributed Transactions library. This helps you differentiate the log messages generated by the MicroTx Distributed Transactions library from the messages generated by the application. Use `[microtx-log]` to filter all the log messages generated by the MicroTx Distributed Transactions library.
 - `%X{microtx-txnId}` refers to the unique transaction ID. When MicroTx Distributed Transactions logs a message, `%X{microtx-txnId}` is replaced with the actual transaction ID.
- b. Provide the names of the MicroTx Distributed Transactions packages which will use the log message definition that you have provided in the previous step.

```

<Logger name="oracle.tmm" level="DEBUG" additivity="false">
    <AppenderRef ref="MicroTxLib"/>
</Logger>

<Logger name="com.oracle.microtx" level="TRACE" additivity="false">
    <AppenderRef ref="MicroTxLib"/>
</Logger>

```

Where,

- `oracle.tmm` and `com.oracle.microtx` are the names of the MicroTx Distributed Transactions packages for which you want the messages to be logged.

- DEBUG and TRACE are the log levels. Enter one of the following values to specify the log level for the MicroTx Distributed Transactions library:
 - INFO: Logs events that occur during the normal operation of the MicroTx Distributed Transactions. This setting logs the least amount of information. This is the default setting.
 - WARNING: Logs events that may cause potentially harmful situations.
 - ERROR: Logs events only if there is a failure. These logs indicate that there is an issue that requires troubleshooting.
 - DEBUG: Logs all the events. Use this setting when you want to debug an issue.
 - OFF: Disables logging so that MicroTx Distributed Transactions logs are not displayed in the MicroTx Distributed Transactions Console for the specified participant service or initiator service.
- MicroTxLib is the value of AppenderRef. This value must match the value of the name that you have entered in the previous step.

The following code provides sample entries for the log4j2.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level
--- [%t] %-50.70C : [microtx-log] %X{microtx-txnId} : %msg%n%throwable"/>
    </Console>

    <Console name="MicroTxLib" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level
--- [%t] %-50.70C : [microtx-log] %X{microtx-txnId} : %msg%n%throwable"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="oracle.tmm" level="DEBUG" additivity="false">
      <AppenderRef ref="MicroTxLib"/>
    </Logger>
    <Logger name="com.oracle.microtx" level="TRACE" additivity="false">
      <AppenderRef ref="MicroTxLib"/>
    </Logger>

    <Root level="info">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

3. For the MicroTx Distributed Transactions Java library for Spring REST applications, create an XML file in your application's resources folder to specify the syntax of the log messages. For example, src/main/resources/logback-spring.xml.
 - a. The following code sample demonstrates how you can define the appender to specify the position of the various fields in the log message.

```
<appender name="ConsoleMTX"
          class="ch.qos.logback.core.ConsoleAppender">
```

```

        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>
                %clr(%d{%LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd
HH:mm:ss.SSS}}){faint} %clr(%LOG_LEVEL_PATTERN:-%5p) %clr(%PID:- )
{magenta} %clr(---){faint} %clr([%15.15t]){faint}
%clr(%-40.40logger{39}){cyan} %clr(:){faint} [microtx-log] %X{microtx-
txnId} : %m%n%LOG_EXCEPTION_CONVERSION_WORD:-%wEx}
            </Pattern>
        </layout>
    </appender>

```

Where,

- *ConsoleMTX* Note down this name as you will provide this name later in the Logger section to define the packages that you want to log.
 - Pattern specifies the syntax of the log messages.
 - [microtx-log] is prefixed to the log messages generated by the MicroTx Distributed Transactions library. This helps you differentiate the log messages generated by the MicroTx Distributed Transactions library from the messages generated by the application. Use [microtx-log] to filter all the log messages generated by the MicroTx Distributed Transactions library.
 - %X{microtx-txnId} refers to the unique transaction ID. When MicroTx Distributed Transactions logs a message, %X{microtx-txnId} is replaced with the actual transaction ID.
- b. Provide the names of the MicroTx Distributed Transactions packages that will use the appender, which you have defined in the previous step.

```

<Logger name="oracle.tmm" level="TRACE" additivity="false">
    <AppenderRef ref="ConsoleMTX"/>
</Logger>
<Logger name="com.oracle.microtx" level="DEBUG" additivity="false">
    <AppenderRef ref="ConsoleMTX"/>
</Logger>

```

Where,

- oracle.tmm and com.oracle.microtx are the names of the MicroTx Distributed Transactions packages for which you want the messages to be logged.
- TRACE and DEBUG are the log levels. Enter one of the following values to specify the log level for the MicroTx Distributed Transactions library:
 - INFO: Logs events that occur during the normal operation of the MicroTx Distributed Transactions. This setting logs the least amount of information. This is the default setting.
 - WARNING: Logs events that may cause potentially harmful situations.
 - ERROR: Logs events only if there is a failure. These logs indicate that there is an issue that requires troubleshooting.
 - DEBUG: Logs all the events. Use this setting when you want to debug an issue.
 - OFF: Disables logging so that MicroTx Distributed Transactions logs are not displayed in the MicroTx Distributed Transactions Console for the specified participant service or initiator service.

- ConsoleMTX is the value of AppenderRef. This value must match the value of the appender name that you have entered in the previous step.

If you want to use another tool to log the information, provide the syntax of the log message in a similar way.

10.4.2 Sample Log Messages

View example log messages and their syntax.

Messages are logged based on the syntax that you have provided in the parser filters as described in [Define the Syntax of Log Messages](#).

Syntax of Logs Generated by the MicroTx Distributed Transactions Coordinator

The MicroTx Distributed Transactions coordinator log files are in JSON format. As shown in the following log snippet, every log generated by the coordinator is associated with a transactionId field. This is the unique ID for a transaction. In case of any failure, you can use this ID to track all the messages related to a transaction. The msg field provides the actual logs associated to the transaction ID.

```
{
  "level": "debug",
  "ts": 1695737654.51672,
  "caller": "xa/server.go:409",
  "msg": "set xa tx expiry timer",
  "transactionId": "9f3efc1c-a831-4d9d-bb53-52337363fcc4"
}
```

Syntax of Logs Generated by the MicroTx Distributed Transactions Java Client Libraries for Helidon Applications

As shown in the following snippet, every log message generated by the MicroTx Distributed Transactions Java client libraries is associated with transactionId. This is the unique ID for a transaction. In case of any failure, you can use this ID to track all the messages related to a transaction.

```
2024-10-30 10:31:57.009 INFO --- [helidon-server-2]
oracle.tmm.jta.common.TrmXAResource : [microtx-log] e9cd912b-323c-4798-91b4-
fa81b096adb1 : new CTX allocate xaOp: PREPARE GTRID: e9cd912b-323c-4798-91b4-
fa81b096adb1 bqual: TCS01-0001 rmid: 4F58D282-0DE1-453C-B872-291FDBF49CFF
```

Where,

- 2024-10-30 10:31:38.454 is the time stamp in YYYY-MM-DD HH:MM:SS.SSS format.
- INFO is the log level.
- helidon-server-2 is the name of the Java application for which the log is generated.
- oracle.tmm.jta.common.TrmXAResource is the name of the class for which the log is generated.
- [microtx-log] is prefixed to the log messages generated by the MicroTx Distributed Transactions library. This helps you differentiate the log messages generated by the MicroTx Distributed Transactions library from the messages generated by the application. Use [microtx-log] to filter all the log messages generated by the MicroTx Distributed Transactions library.

- e9cd912b-323c-4798-91b4-fa81b096adb1 is the unique transaction ID.
- new CTX allocate xaOp: PREPARE GTRID: e9cd912b-323c-4798-91b4-fa81b096adb1 bqual: TCS01-0001 rmid: 4F58D282-0DE1-453C-B872-291FDBF49CFF is the actual log message associated with the transaction ID.

You can use Mapped Diagnostic Context (MDC) key to change the position of the transaction ID in the logs. See [Define the Syntax of Log Messages](#).

Syntax of Logs Generated by the MicroTx Distributed Transactions Java Client Libraries for Spring REST Applications

As shown in the following snippet, every log message generated by the MicroTx Distributed Transactions Java client libraries for a Spring REST application is associated with transactionId. This is the unique ID for a transaction. In case of any failure, you can use this ID to track all the messages related to a transaction.

```
2024-10-30 10:31:56.785 DEBUG 30201 --- [nio-8082-exec-3]
c.o.microtx.common.HeaderPropagation : [microtx-log] e9cd912b-323c-4798-91b4-
fa81b096adb1 : Authz Header Propagated: oracle-tmm-tx-token
```

Where,

- 2024-10-30 10:31:56.785 is the time stamp in YYYY-MM-DD HH:MM:SS.SSS format.
- DEBUG is the log level.
- nio-8082-exec-3 is the name of the Spring REST application for which the log is generated.
- c.o.microtx.common.HeaderPropagation is the name of the class for which the log is generated.
- [microtx-log] is prefixed to the log messages generated by the MicroTx Distributed Transactions library. This helps you differentiate the log messages generated by the MicroTx Distributed Transactions library from the messages generated by the application. Use [microtx-log] to filter all the log messages generated by the MicroTx Distributed Transactions library.
- e9cd912b-323c-4798-91b4-fa81b096adb1 is the unique transaction ID.
- Authz Header Propagated: oracle-tmm-tx-token is the actual log message associated with the transaction ID.

You can use Mapped Diagnostic Context (MDC) key to change the position of the transaction ID in the logs. See [Define the Syntax of Log Messages](#).

Syntax of Logs for MicroTx Distributed Transactions Node.js Client Libraries

As shown in the following snippet, every log message generated by the MicroTx Distributed Transactions Node.js client libraries is prefixed with the transaction ID. This is the unique ID for a transaction. In case of any failure, you can use this ID to track all the messages related to a transaction.

```
2023-09-28 6:53:56.678 :: edc36bc0-823e-4660-886f-d7e67c97aa80 - Service
Enlisted with LRA : http://localhost:9000/api/v1/lra-coordinator/
edc36bc0-823e-4660-886f-d7e67c97aa80
```

Where,

- 2023-09-28 6:53:56.678 is the time stamp in YYYY-MM-DD HH:MM:SS.SSS format.
- edc36bc0-823e-4660-886f-d7e67c97aa80 is the unique ID of the transaction.
- Service Enlisted with LRA : http://localhost:9000/api/v1/lra-coordinator/edc36bc0-823e-4660-886f-d7e67c97aa80 is the message.

10.4.3 Enable Logs for MicroTx Distributed Transactions Node.js Library

By default, logging is enabled for the MicroTx Distributed Transactions Coordinator and Java applications that use the MicroTx Distributed Transactions client library. However, for Node.js applications that use the MicroTx Distributed Transactions client libraries, you must enable logging.

In the `tmm.properties` file, where you have provided values to configure the MicroTx Distributed Transactions library, specify values for the following properties to enable logging for the Node.js application:

1. Set `oracle.tmm.EnableDebugLogs` as `true`.

The default value is `false`. If you set this value to `false`, MicroTx Distributed Transactions client libraries logs messages only when there is an error.

2. Set `oracle.tmm.enableLogTimestamp` as `true` if you want the the time stamp to be displayed in YYYY-MM-DD HH:MM:SS.SSS format.

The default value is `false`, so the time stamp is not displayed by default.

You can also provide these configuration values as environment variables. Note that if you specify values in both the `tmm.properties` file as well as the environment variables, then the values set in the environment variables override the values in the properties file.

The following example provides sample values to configure the environment variables.

```
export ENABLE_DEBUG_LOGS=true
export ENABLE_LOG_TIMESTAMP=true
```

Note that the environment variables names are case-sensitive.

11

Troubleshooting

Learn about the issues you may encounter and how to work around them. The following notifications may appear in the Web Console.

Topics:

- [Cannot retrieve transaction details from the MicroTx Distributed Transactions coordinator](#)
Check the configuration details provided for coordinator and data store, and then redeploy the MicroTx Distributed Transactions coordinator.
- [Unable to retrieve details about the MicroTx Distributed Transactions coordinator and console health](#)
Check if the Kubernetes Metrics Server is deployed.
- [No data is displayed in the Console](#)
When the Health service is not reachable and no data is displayed in the MicroTx Distributed Transactions console, refresh the MicroTx Distributed Transactions Console to resolve this issue.
- [Handle a spike in traffic for Helidon apps](#)
When there is an increase in the load of threads that a Helidon application processes, increase the server thread pool size so that the Helidon application can handle the traffic.

11.1 Cannot retrieve transaction details from the MicroTx Distributed Transactions coordinator

Check the configuration details provided for coordinator and data store, and then redeploy the MicroTx Distributed Transactions coordinator.

Cause

An attempt was made to retrieve transaction details from the MicroTx Distributed Transactions coordinator, but the details could not be retrieved.

Action

Identify the issue, fix it, and then deploy the MicroTx Distributed Transactions coordinator.

- If you are using etcd or Oracle Database as the data store, check that the data store is running and connected to the MicroTx Distributed Transactions coordinator.
- If you are using an Oracle Database as the data store, ensure that you have granted the required privilege to the user. See [Grant Privileges to Database User](#).
- Check that you have provided the correct configuration details for the MicroTx Distributed Transactions coordinator in the `values.yaml` file. See [Configure the values.yaml File](#).
- Check if the network is available and network traffic is permitted on the ports.

11.2 Unable to retrieve details about the MicroTx Distributed Transactions coordinator and console health

Check if the Kubernetes Metrics Server is deployed.

Cause

The Metrics Server collects the resource metrics that indicate the current health status of the MicroTx Distributed Transactions coordinator and console. This server is not reachable.

Action

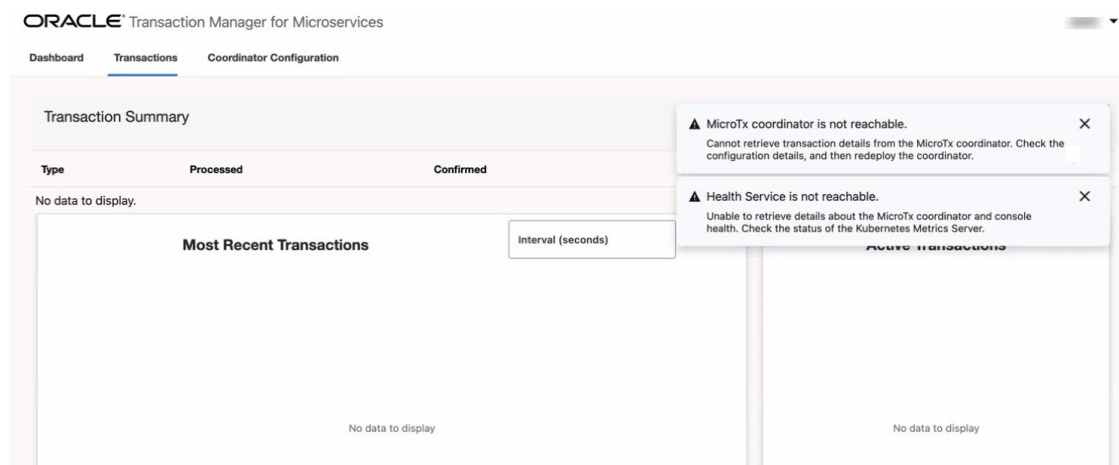
Redeploy the Metrics Server. See Deploy Kubernetes Metrics Server. Also verify that the Kubernetes cluster is running.

11.3 No data is displayed in the Console

When the Health service is not reachable and no data is displayed in the MicroTx Distributed Transactions console, refresh the MicroTx Distributed Transactions Console to resolve this issue.

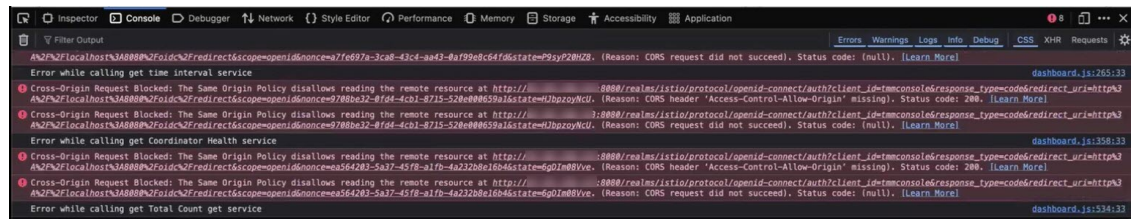
Cause

The following error messages are displayed when the MicroTx Distributed Transactions Console is left idle for along time and the authentication token expires.



As shown in the image, error messages are displayed to indicate that the health service and MicroTx Distributed Transactions coordinator are not reachable. Additionally, you cannot navigate from one tab to another. No data is displayed in the MicroTx Distributed Transactions console when you try to navigate between tabs.

This issue occurs when MicroTx Distributed Transactions Console makes an attempt to access the authentication token, but the token has expired. To confirm if the authentication token has expired and to debug, open developer tools in your browser and view the messages that appear in the **Console** tab. As shown in the following figure, the Cross-Origin Request Blocked error message is displayed when the authentication token expires.



Action

To resolve this issue, perform one of the following actions:

- Refresh the MicroTx Distributed Transactions Console.
- Increase the lifespan of the authentication token. Refer to the documentation of your identity provider for details on how to increase the lifespan of the token.

11.4 Handle a spike in traffic for Helidon apps

When there is an increase in the load of threads that a Helidon application processes, increase the server thread pool size so that the Helidon application can handle the traffic.

Cause

When you run Helidon applications with multiple replicas and a heavy load of multiple threads, ensure that the pool size is adequate to handle the traffic. If there is a spike in traffic while the pool size is small, then requests will either get queued or dropped.

Action

To resolve this issue, specify values for the following properties. These properties are specific to Helidon applications and are not a part of MicroTx Distributed Transactions library properties.

- **core-pool-size:** specifies the minimum number of threads that are always kept alive.
- **max-pool-size:** specifies the maximum number of threads that can be created to handle the load.

You can set these property values either in the Helm charts or in the `application.properties` file, which is specific to each application.

Sample entry in Helm Chart

The following example specifies property values under the environment variable section for a Helidon application.

```
env:
  - name: SERVER_EXECUTOR_SERVICE_CORE_POOL_SIZE
    value: "200"
  - name: SERVER_EXECUTOR_SERVICE_CORE_POOL_SIZE
    value: "500"
```

Sample entry in the application.properties file

The following example specifies property values for a Helidon application in the `tmm.properties` file.

```
server.executor-service.core-pool-size: 200  
server.executor-service.max-pool-size: 500
```

A

Deploy Your Application

Before you begin, ensure that you have completed the following tasks:

1. Installed Transaction Manager for Microservices - Distributed Transactions (MicroTx Distributed Transactions).
2. Integrated MicroTx Distributed Transactions library with your application code.

Workflow to deploy your application

Build your application code to create a Docker image for each application microservice, push the Docker image to a remote repository, set up the required environment, enter the configuration details in the YAML file, and then install the application.

1. [Build the Docker Image](#)
2. [Push App Image to a Remote Repo](#)
3. [Create Helm Files](#)
4. [Install Your Application](#)

- [Build the Docker Image](#)

Your application may consist of multiple microservices. Build the source code for each microservice, so that you create an image for each microservice. For each microservice, run the command discussed in this section from the root folder of the microservice for building the Docker image.

- [Push App Image to a Remote Repo](#)

Push the Docker image of the applications, that you have built, to a remote repository.

- [Create Helm Files](#)

After integrating the MicroTx Distributed Transactions libraries with your application code, you can install the application.

- [Install Your Application](#)

A.1 Build the Docker Image

Your application may consist of multiple microservices. Build the source code for each microservice, so that you create an image for each microservice. For each microservice, run the command discussed in this section from the root folder of the microservice for building the Docker image.

1. Build the application source code to create a container image.

Use the following command to create a container image with the tag `local_image_tag`.

```
docker build -t local_image_tag .
```

When you run this command in your environment, you can specify any tag that you want after the `-t` option.

2. Note down the tag that you have associated with this image. You will need to specify this tag later.

The container image that you have created is available in your local Docker container registry.

A.2 Push App Image to a Remote Repo

Push the Docker image of the applications, that you have built, to a remote repository.

The container image that you have built is available in your local repository. You must push this image to a remote repository, so that you can access this image using Helm. Later, you will use Helm to install your application.

If you are using Oracle Cloud Infrastructure Registry, see [Push an Image to Oracle Cloud Infrastructure Registry](#). If you are using other Kubernetes platforms, use the instructions provided in this section.

Before you begin, complete the following tasks:

- Identify a remote private repository to which you want to upload the container image. You can create a new remote Docker repository or use an existing one. Use a private repository to limit access. When you use a remote Docker repository, you have to push images to the remote Docker repository only once, while you can pull an image multiple times onto any Kubernetes cluster that you create.
- Create a Kubernetes secret to access the remote Docker repository. See [Create a Kubernetes Secret to Access Docker Registry](#).
- 1. Provide credentials to log in to the remote private repository to which you want to push the image.

```
docker login <repo>
```

Provide the login credentials based on the Kubernetes platform that you are using.

2. In your local container registry, identify the tag of the image that you want to push.

Skip this step if you have noted the tag of this image.

- a. Run the following command to list the Docker images.

```
docker images
```

- b. Copy the tag of the image that you want to push. You'll need to provide this information later.
3. Use the following command to specify a unique tag for the image that you want to push to the remote Docker repository.

Syntax

```
docker tag local_image_tag remote_image_tag
```

Where,

- *local_image_tag* is the tag with which the image is identified in your local repository.
- *remote_image_tag* is the tag with which you want to identify the image in the remote Docker repository.

Example Command

```
docker tag myApp123 <region-key>.ocir.io/otmmrepo/myApp123
```

Where, <region-key>.ocir.io/otmmrepo is the Oracle Cloud Infrastructure Registry to which you want to push the image file, myApp123. If you are using other Kubernetes platforms, then provide the registry details based on your environment.

4. Push the Docker image from your local repository to the remote Docker repository.

Syntax

```
docker push remote_image_tag
```

Example Command

```
docker push <region-key>.ocir.io/otmmrepo/myApp123
```

Note down the tag of the Docker image in the remote Docker repository. You'll need to enter this tag while pulling the image from the remote Docker repository.

A.3 Create Helm Files

After integrating the MicroTx Distributed Transactions libraries with your application code, you can install the application.

If you want to use Helm to install the application, you must create the required YAML files and charts. Each sample in the installation bundle contains the sample application code and the required YAML files to install the sample application using Helm. Use these files as a reference to create files for your application.

A.4 Install Your Application

1. Navigate to the folder that contains the Helm files for your application.
2. Deploy your application using the configuration details that you have provided in the values.yaml file.

Syntax

```
helm install <release name> --namespace <namespace> <chart directory> --values <values.yaml>
```

Example

Use the following commands to install your application with the name my-java-app-tcc-tx in the MicroTx namespace.

```
helm install my-java-app-tcc-tx --namespace MicroTx .\ --values .\values.yaml
```

Where,

- my-java-app-tcc-tx is the name of the application that you want to create.
- MicroTx is the namespace in Kubernetes cluster, where you want to install your application.
- .\ is the folder that contains the chart.yaml file for your application. Since, you have already changed the directory to the helmchart folder on the command-line, you can provide the relative path to the chart.yaml file.

- `.\values.yaml` is the location of the `values.yaml` file, the application's manifest file, in your local machine. This file contains the deployment configuration details for your application.
3. Verify that all resources, such as pods and services, are ready. Use the following command to retrieve the list of resources in the namespace `MicroTx` and their status.

```
kubectl get all -n MicroTx
```

4. Verify that the application is installed.

```
helm list --namespace MicroTx
```