Oracle®

Transaction Manager for Microservices Developer Guide

Release 24.2
F48194-14
June 2024

Oracle Transaction Manager for Microservices Developer Guide, Release 24.2

F48194-14

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

Contents

C Run MicroTx in a Docker Container

What's New in MicroTx

The following are the changes in the Transaction Manager for Microservices (MicroTx) release 23.4.1 and the previous releases.

- New Features in Release 24.2
  The following features were introduced in MicroTx release 24.2.
- Changes in the Previous Release
  The following are the changes in previous releases of MicroTx.

New Features in Release 24.2

The following features were introduced in MicroTx release 24.2.

Enhancements in the MicroTx Library

The enhanced MicroTx library provides functionality for the following applications:

- Spring REST-based applications can initiate a new TCC transaction or participate in an existing TCC transaction with MicroTx. See Develop Spring REST Apps with TCC.
- Micronaut applications can initiate a new Saga transaction or participate in an existing Saga transaction. See Develop Micronaut Apps with Saga.
- Oracle Database applications, built using Oracle APEX and Oracle REST Data Services (ORDS), can now initiate a new XA transaction with MicroTx by using the MicroTx PL/SQL library. See Develop ORDS App with XA.
- JAX-RS and Spring REST-based applications can participate in an XA transaction with MicroTx while leveraging the Oracle Transactional Event Queues (TEQ) feature available in Oracle Database. See Configure Java Apps to Leverage Transactional Event Queues.
- Spring Boot applications using Spring REST and Micronaut applications can initiate or participate in a Saga transaction with MicroTx while supporting auto-compensation using the lock-free reservation feature available in Oracle Database 23ai. See Develop Java Apps that Use Saga and Lock-Free Reservation.

Send Notifications Using Prometheus Alertmanager

Install and configure Prometheus Alertmanager to send notifications based on specific thresholds and rules. For example, users are notified when the MicroTx coordinator service or the database service is not available. See Install and Configure Alertmanager.

Enhanced Troubleshooting through Console Notifications

Notifications appear on the MicroTx web console in case of certain failures. These messages help to identify and troubleshoot the issue. See Troubleshooting.

MicroTx is Now Available with OpenShift 4.14.x

In addition to deploying MicroTx on Kubernetes cluster or Docker Swarm, you can now deploy MicroTx in OpenShift 4.14.x or a compatible version. See Supported Container Platforms.

Sample Helm Charts to Install MicroTx Without Istio Service Mesh

In Kubernetes, you can install MicroTx 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 on a Kubernetes cluster with and without using the Istio service mesh. See Prepare a Kubernetes Cluster.

Changes in the Previous Release

The following are the changes in previous releases of MicroTx.

- Changes in 23.4.2
  Experience enhanced performance and resilience with MicroTx Free release 23.4.2.
- Changes in the Enterprise Edition Release 23.4.1
  The following features were introduced in MicroTx Enterprise Edition release 23.4.1.
- Changes in Free Release 23.4.1
  The following features were introduced in MicroTx Free release 23.4.1.
- Changes in 22.3.2
  The following new features were introduced in MicroTx 22.3.2.
- Changes in 22.3.1
  The following new feature was introduced in MicroTx release 22.3.1.

Changes in 23.4.2

Experience enhanced performance and resilience with MicroTx Free release 23.4.2.

Changes in the Enterprise Edition Release 23.4.1

The following features were introduced in MicroTx Enterprise Edition release 23.4.1.

Manage Transactions Using the MicroTx Console

View the health of all the replicas of the MicroTx transaction coordinator and manage transactions using an easy-to-use graphical web console. See Manage Transactions Using the Web Console.

Manage Transaction Promotion

The MicroTx client library manages the local transactions. By handling a transaction locally, you can experience better performance. Local transactions save time and increase throughput as the MicroTx coordinator does not read, write, coordinate, or save database cache and logs for such transactions. See About Global and Local Transactions.

Visualize Metrics Using Grafana

The MicroTx coordinator exposes metrics in a format that can be easily read by Prometheus. Use Grafana to visualize the metrics data collected into Prometheus. You can import MicroTx dashboards to view the metrics in Grafana. Use these metrics to monitor your transactions and health of the MicroTx coordinator. See Monitor Performance.

MicroTx Supports Oracle RAC as a Resource Manager

You can use Oracle Real Application Clusters (RAC) as a resource manager for Node.js, JAX-RS, and Spring REST applications use the XA transaction protocol. See Supported Resource Managers.

Caching the Transaction Logs

Caching the transaction logs that are stored in etcd or Oracle Database to improve performance and optimize the read and write operations. See Enable Caching.

Create Multiple Replicas of the Transaction Coordinator

You can run multiple replicas of Transaction Manager for Microservices pod at a time. Oracle recommends a minimum of 3 replicas for production environments. See Environment Details.

You can scale up or down the number of replicas based on the number of transactions. When the number of transaction requests is low, scale down the number of replicas to use the resources efficiently.

Optimize Transactions that Use a Common Resource Manager

Based on your business requirements, you may use a single resource manager for multiple transaction participant services. 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. See Common Resource Manager for Multiple Apps.

Store Transaction Details in Oracle Database or etcd

In addition to internal memory, MicroTx now supports etcd or Oracle Database as a data store for persistence of transaction state. See Supported Data Stores for Transaction Logs.

Recover Transactions

In case the transaction coordinator server fails, MicroTx resumes the transactions that are in progress after the server restarts. See About Transaction Recovery.

No Limit on the Number of Transactions

You can run up to 4800 transactions per hour across all the transaction protocols and across all replicas of the transaction coordinator when you use Transaction Manager for Microservices Free. There is no limit on the number of transactions in Transaction Manager for Microservices Enterprise Edition.

Changes in Free Release 23.4.1

The following features were introduced in MicroTx Free release 23.4.1.

Enhancements in the MicroTx Library

The enhanced MicroTx library provides functionality for the following applications:

- Helidon 3.x in Jakarta EE9 environment can initiate a new XA transaction or participate in an existing XA transaction. See Develop JAX-RS Apps with XA.
- Spring REST-based applications can initiate a new XA and Saga transactions or to participate in an existing XA and Saga transaction. See Develop Spring REST-based Applications with Saga and Develop Spring REST Apps with XA.

MicroTx Supports Use of the `@Transactional` Annotation

You can easily configure your Java applications that use the XA transaction protocol by using the `@Transactional` annotation. See About @Transactional.

Enhanced Diagnostics and Troubleshooting through Log Correlation

The MicroTx coordinator and the MicroTx client libraries generate logs. In case of any failure, view the logs to identify the issue and troubleshoot. See Logs.

Changes in 22.3.2

The following new features were introduced in MicroTx 22.3.2.

Last Resource Commit (LRC) Optimization for XA Transactions

In addition to Logging Last Resource (LLR) optimization, you can now use Last Resource Commit (LRC) optimization to enable one non-XA resource to participate in a global XA transaction. See Optimizations for a Non-XA Resource.

Support for Multiple Resource Managers for a Service

Based on your application's business logic, you can use multiple resource managers for a single participant service. A participant service can connect to multiple XA-compliant resource managers. However, only one non-XA resource is supported in a transaction. See Configure Multiple Resource Managers for a Single App.

MicroTx Library for Python Apps Using the TCC Transaction Protocol

The MicroTx library for Python provides the functionality to Python applications to initiate a new TCC transaction or to participate in an existing TCC transaction. Earlier, the MicroTx library for TCC transaction protocol supported only Java and Node.js applications. See Develop Python Apps with TCC.

Subscribe to Receive XA Transaction Notifications

You can register your transaction initiator and participant services to receive notifications. MicroTx notifies the registered services when the following events occur: before the prepare phase and when MicroTx successfully commits or rolls back a transaction. You may want to register your service, if based on the business logic your service performs additional tasks when an event occurs. See Subscribe to Receive XA Transaction Notifications.

MicroTx Library for WebLogic Server J2EE Applications

You can integrate the MicroTx library with your WebLogic Server J2EE applications. See Integrating XA Global Transactions Between WebLogic Server and Helidon Using MicroTx in *Integrating Oracle WebLogic Server with Helidon*.

Changes in 22.3.1

The following new feature was introduced in MicroTx release 22.3.1.

Support for Session Affinity

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 unique transaction or session are routed to the same endpoint or replica of the participant service that served the first request. Depending on your business use case, you may have to enable session affinity for the transaction participant service or the transaction coordinator. See About Session Affinity.

1
About MicroTx

Oracle Transaction Manager for Microservices (MicroTx) 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 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.

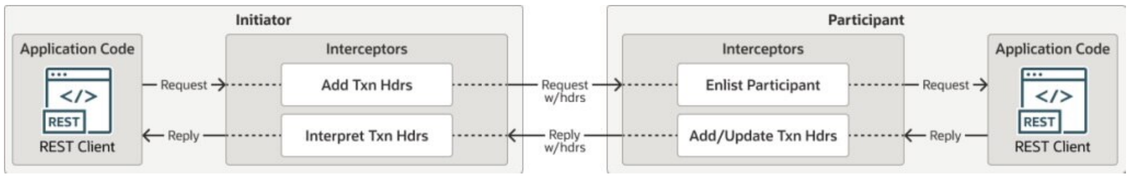
1.1 How MicroTx Works

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

About interceptors provided by the MicroTx client libraries

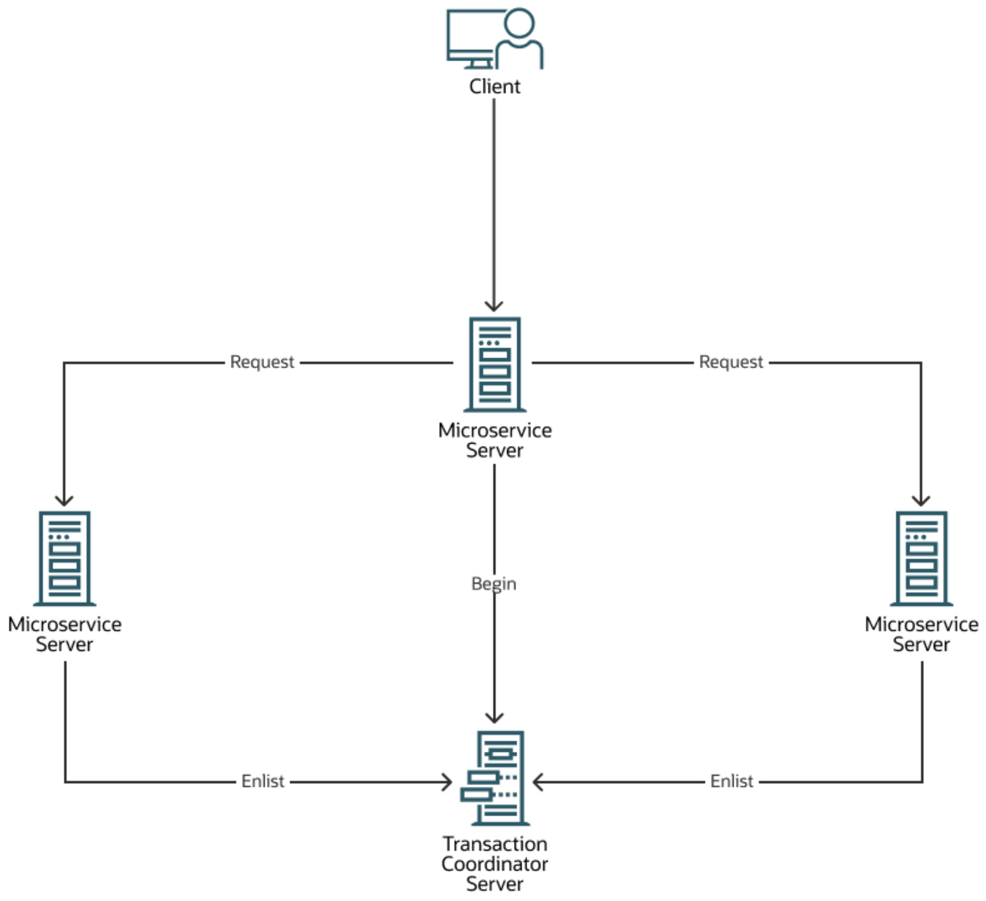
The MicroTx client libraries provide interceptors to intercept both incoming and outgoing REST calls, as well as their requests and responses. These interceptors use headers to propagate the transaction context which enable the participant microservices to automatically enlist in a transaction. The interceptors 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 libraries.



When a microservice, that uses the MicroTx 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. The following figure shows how MicroTx communicates with your application microservices to handle transactions.

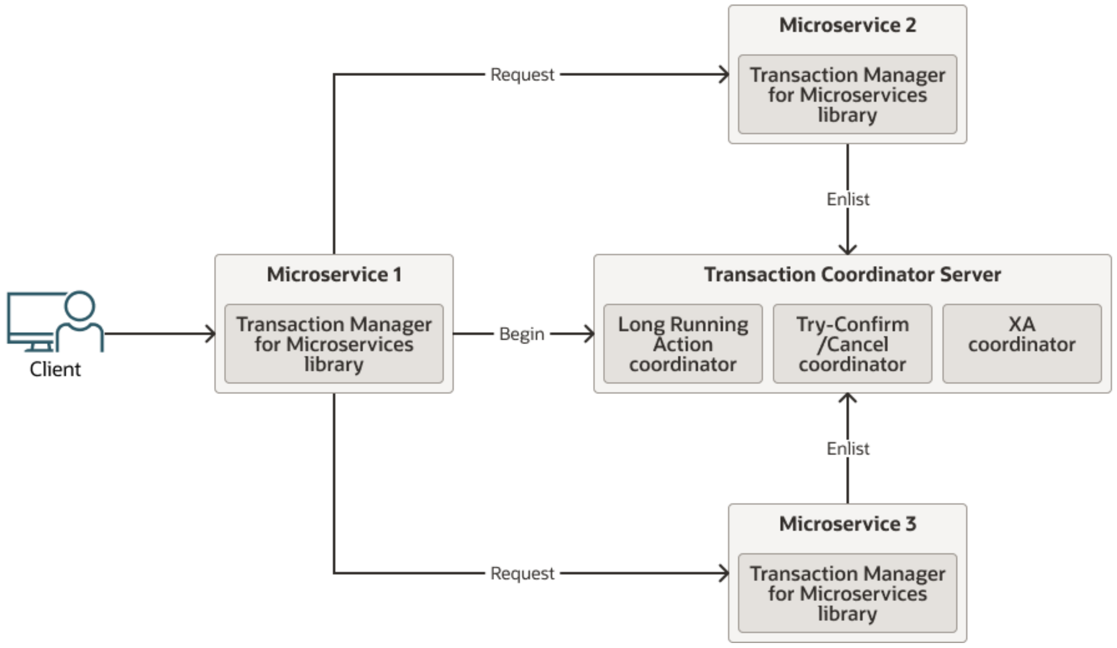


1. Application developers use functions present in the MicroTx library with their application code.
2. When a microservice or client initiates a transaction, it calls functions in the MicroTx library to start a distributed transaction.
3. MicroTx library includes headers that enable the participant services to automatically enlist in the transaction.
4. After all the tasks associated 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

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

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


Microservice 2
Transaction Manager for Microservices library
Request
Enlist
Microservice 1
Transaction Manager for Microservices library
Client
Begin
Transaction Coordinator Server
Long Running Action coordinator
Try-Confirm /Cancel coordinator
XA coordinator
Enlist
Microservice 3
Transaction Manager for Microservices library
Request

Transaction Coordinator Server

The transaction coordinator manages transactions amongst the participant services.

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

MicroTx library

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

Application developers use different parts of the MicroTx 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 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 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

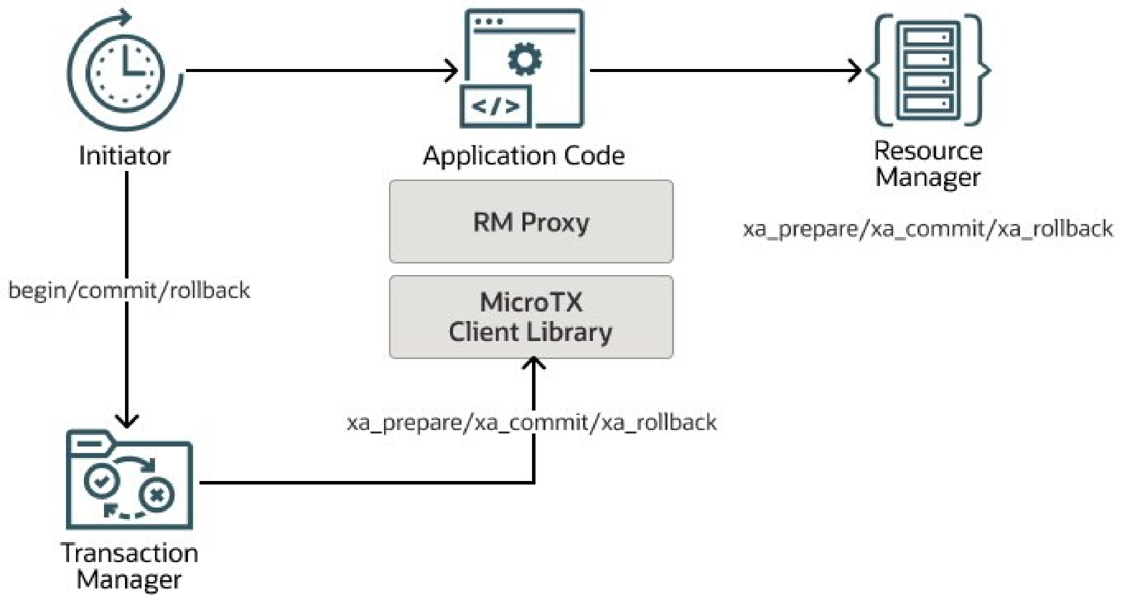
Use XA when strong consistency is required, similar to consistency provided by the local database transactions, where all the ACID properties of a transaction are present. For example, financial applications. Use the Saga protocol for transactions that may take a long time to complete. You can use the 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 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 commits or rolls back the transaction.

In the XA protocol, participant microservices must use the MicroTx client libraries which registers callbacks and provides implementation of the callbacks for the resource manager. As shown in the following image, MicroTx communicates with the resource managers to commit or roll back the transaction. MicroTx 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 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.

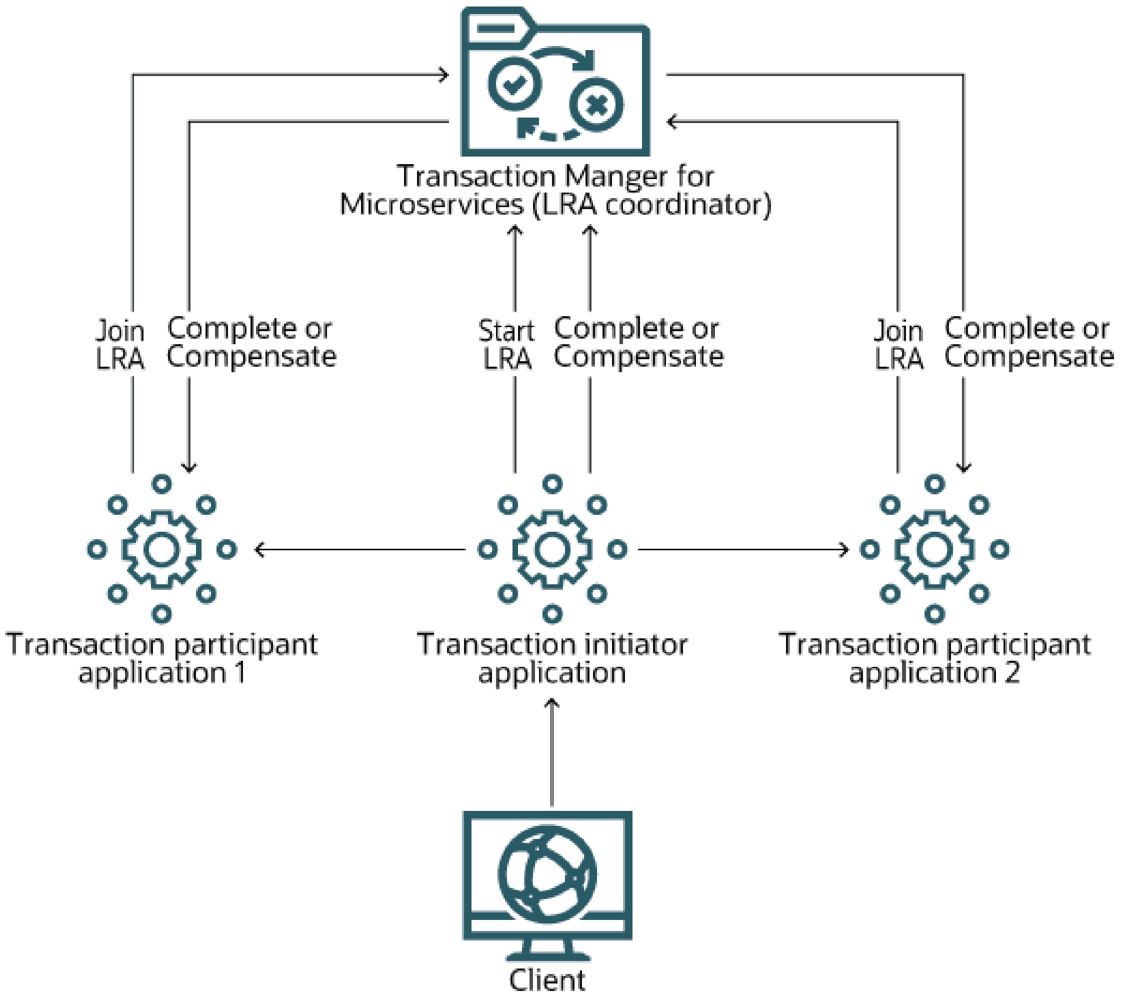


1. Initiator starts the distributed transaction
2. Called microservices enlist in the transaction
3. Initiator asks transaction manager to commit or rollback the transaction
4. If the initiator decided to commit, the transaction manager asks each microservice to prepare
   a. If all participants successfully 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 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 when you use the Saga transaction protocol, which is based on Eclipse MicroProfile Long Running Actions (LRA).


Transaction Manger for Microservices (LRA coordinator)
Join LRA
Complete or Compensate
Start LRA
Complete or Compensate
Join LRA
Complete or Compensate
Transaction participant application 1
Transaction initiator application
Transaction participant application 2
Client

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

1. The transaction initiator service calls the MicroTx 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 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 to either complete or compensate the transaction.
5. MicroTx 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.

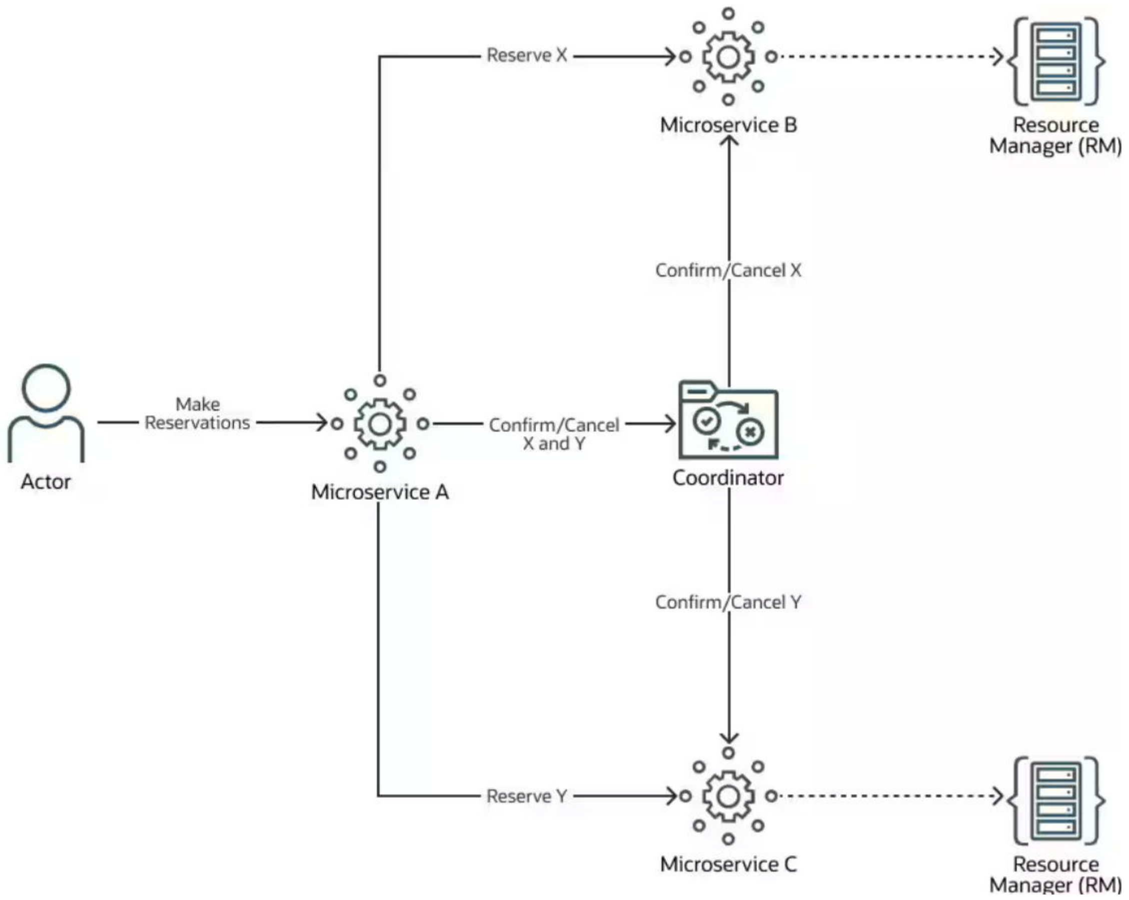
1.3.3 Try-Confirm/Cancel Transaction Protocol

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 client libraries places the URI in MicroTx 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 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 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 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 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 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. In case of failures, the initiator service calls cancel instead of confirm.

1. The transaction initiator service, Microservice A, makes a MicroTx 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 to either confirm or cancel the reservations.
7. MicroTx 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.
9. MicroTx returns a successful status to the transaction initiator, Microservice A. If MicroTx does not receive 200 status from one or more participants, then it returns an error message.

1.4 Workflow to Install and Use MicroTx

Use the following workflow as a guide to install, configure, and use MicroTx to manage transactions.

Task	Description	See
Understand the requirements and select a transaction protocol for your application	Plan the installation and setup of MicroTx based on your business requirements.	Plan
Download the installation bundle	The installation bundle contains the MicroTx 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 image to Docker registry, provide configuration information, and then install MicroTx.	You can install MicroTx on a Kubernetes cluster or Docker Swarm. Provide configuration information in the `values.yaml` file for Kubernetes and the `tcs-docker-swarm.yaml` file for Docker Swarm.	Install on a Kubernetes Cluster or Install on Docker Swarm
Access MicroTx	Verify that MicroTx was installed properly and access the service.	Post-Installation Tasks
Run Sample Applications	Optional. Using samples is the fastest way for you to get familiar with MicroTx.	Deploy Sample Applications
Use MicroTx 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.	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 Plan

Consider the points discussed in this section to plan the installation and setup of Transaction Manager for Microservices (MicroTx).

- Supported Container Platforms
- Supported Authorized Cloud Environments
- Supported Languages and Frameworks
- Supported Data Stores for Transaction Logs
  To persist the transaction logs, MicroTx uses a data store.
- Supported Identity Providers
- Limits
  MicroTx permits 4800 transactions per hour across all the transaction protocols and across all replicas of the transaction coordinator.
- About Transaction Protocols
  Use the information provided to select a transaction protocol for your application based on your business requirements.
- About Authentication and Authorization
  Authentication ensures that only authorized individuals can access the Transaction Manager for Microservices (MicroTx) coordinator, the microservices, transaction, and data. Authorization provides access control to system privileges and data. This builds on authentication to ensure that individuals get appropriate access.

2.1 Supported Container Platforms

You can deploy MicroTx on Docker or Kubernetes cluster. MicroTx is tested with the following platforms:

- Kubernetes 1.21.x. Use any Kubernetes distribution that is compatible with Kubernetes 1.21.x.
- Docker 20.10.x. Use any operating system that supports Docker 20.10.x or a compatible version.
- OpenShift 4.14.x. Use any operating system that supports OpenShift 4.14.x or a compatible version.

2.2 Supported Authorized Cloud Environments

You can use the MicroTx Enterprise Edition licensing in the following authorized cloud environments.

- Amazon Web Services – Kubernetes clusters on Amazon Elastic Compute Cloud (EC2) and Amazon Relational Database Service (RDS)

- Microsoft Azure Platform - Kubernetes clusters on Azure virtual machines, Azure Kubernetes Services (AKS), Azure Arc, and Azure Stack

For more information about licensing information, see https://www.oracle.com/a/ocom/docs/corporate/oracle-software-licensing-basics.pdf.

2.3 Supported Languages and Frameworks

Use MicroTx to ensure transactional consistency across microservices application implemented in the following languages:

- TypeScript or JavaScript for Node.js
- Java 11 (for applications built with frameworks, such as Helidon 2.x and WebLogic Server)
- Java 17 (for applications built with frameworks, such as Helidon 3.x, Helidon 4.x, Spring Boot 3.x, and Micronaut 4.2.1 or later)
- Python 3.11 or later

MicroTx supports Node.js and Java for all the transaction protocols and supports Python only for TCC.

Supported Java Frameworks

The MicroTx libraries are available for Spring REST applications and JAX-RS applications that use the XA, Saga, and TCC transaction protocols. MicroTx library is also available for Micronaut applications that use the Saga transaction protocol and ORDS applications that use the XA transaction protocol.

The MicroTx XA library is available for the following Java frameworks:

- Spring Boot 3.x, which includes Hibernate and EclipseLink applications
- Helidon 2.x, 3.x, and 4.x
- Oracle WebLogic Server 14. See Integrating XA Global Transactions Between WebLogic Server and Helidon Using MicroTx in *Integrating Oracle WebLogic Server with Helidon*.
- Oracle Tuxedo 22c
- Oracle REST Data Services (ORDS) 19c
- Micronaut 4.2.1 or later

2.4 Supported Data Stores for Transaction Logs

To persist the transaction logs, MicroTx uses a data store.

MicroTx supports the following data stores.

- etcd
- Oracle Database running in these environments:
  - All supported versions of Oracle On-Premise Database
  - Autonomous Database for Transaction Processing and Mixed Workloads - both shared and dedicated
  - Bare Metal and Virtual Machine DB Systems in Oracle Cloud Infrastructure
  - Oracle Exadata Cloud Service
  - Oracle Exadata Cloud@Customer

  – Oracle Real Application Clusters (RAC) 19c

  You can connect to an Oracle Database in your on-premises environment or connect to an Oracle Cloud Infrastructure Database service.

2.5 Supported Identity Providers

You can use the following identity providers to create the authentication information and secure communication.

- Oracle IDCS
- Oracle IAM
- Keycloak
- Microsoft Azure Active Directory and Active Directory

This guide provides information about creating an access token using Oracle IAM and Oracle IDCS.

If you want to use Keycloak or Microsoft AD as the identity provider, refer to their product documentation for information about setting up the identity provider and creating an access token.

2.6 Limits

MicroTx permits 4800 transactions per hour across all the transaction protocols and across all replicas of the transaction coordinator.

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.

This limit applies only to the Transaction Manager for Microservices Free release. In Transaction Manager for Microservices Enterprise Edition, there is no limit on the number of transactions.

2.7 About Transaction Protocols

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

Parameters	XA	Saga	TCC
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.

2.8 About Authentication and Authorization

Authentication ensures that only authorized individuals can access the Transaction Manager for Microservices (MicroTx) coordinator, the microservices, transaction, and data. Authorization provides access control to system privileges and data. This builds on authentication to ensure that individuals get appropriate access.

- About Access and Refresh Tokens
  Use access and refresh tokens to ensure that only authenticated users can access the service and to permit only the administrative user or the user that originally initiated the transaction to manage a transaction.
- About the Oracle_Tmm_Tx_Token Transaction Token
  Enable the creation and propagation of the transaction token to ensure that only authorized users have access to the service. When you set `transactionTokenEnabled` to `true` in the YAML file, MicroTx creates a new token called `Oracle_Tmm_Tx_Token`, which is a signed transaction token.
- About Encrypting and Storing Tokens
  To support asynchronous calls, MicroTx stores the access and refresh tokens, and then uses it in asynchronous calls.

2.8.1 About Access and Refresh Tokens

Use access and refresh tokens to ensure that only authenticated users can access the service and to permit only the administrative user or the user that originally initiated the transaction to manage a transaction.

Use an identity provider to create an access token and a refresh token. When you send a new REST API request, such as a request to book a trip, you must pass the access and refresh tokens in the request header.

Access Token

When you enable authentication, you must pass the access token in the `authorization` header with every request. MicroTx enforces JWT-based authentication and validates the access token in all incoming requests against the public key. It also validates all the calls sent from the MicroTx library to the transaction coordinator. MicroTx checks that the user who passes the access token has the required system privileges to perform the operation. This ensures that only authorized users can access the MicroTx APIs.

When you enable authorization checks at coordinator and if you do not provide the access token when you send the request, the transaction is rejected as there is no access token.

Refresh Token

Refresh token is used to refresh an expired access token. Asynchronous calls or transactions could span a few minutes or hours. For example, you use the Saga transaction protocol to book a hotel and flight. It can take a few minutes for the user to complete the bookings. However, the access token could expire before the user completes the transaction. When you specify the URL and client ID of the identity provider in the YAML file, MicroTx provides the refresh token to the identity provider and gets a new access token.

2.8.2 About the Oracle_Tmm_Tx_Token Transaction Token

Enable the creation and propagation of the transaction token to ensure that only authorized users have access to the service. When you set `transactionTokenEnabled` to `true` in the YAML file, MicroTx creates a new token called `Oracle_Tmm_Tx_Token`, which is a signed transaction token.

The following steps describe how MicroTx creates the `Oracle_Tmm_Tx_Token` transaction token and propagates it in the subsequent communication between the participant services and MicroTx.

1. When a user begins a transaction, the transaction initiator service sends a request to MicroTx.
2. MicroTx responds to the transaction initiator and returns `Oracle_Tmm_Tx_Token` in the response header.
   The MicroTx library creates this token based on the public-private 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 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. For all the subsequent calls from the participant services to the transaction coordinator, the MicroTx library passes `Oracle_Tmm_Tx_Token` in the request header.

To enable propagation of the transaction token in a Kubernetes Cluster, see Transaction Token Properties.

To enable propagation of the transaction token in Docker Swarm, see Transaction Token Properties.

2.8.3 About Encrypting and Storing Tokens

To support asynchronous calls, MicroTx stores the access and refresh tokens, and then uses it in asynchronous calls.

To encrypt the tokens, create encryption keys. MicroTx encrypts the tokens and stores it. When there is an asynchronous call from MicroTx to a participant service, MicroTx fetches the encrypted token, decrypts it, and then attaches the token to the request header.

MicroTx encrypts the access and refresh tokens, and then uses it later while making calls to participant services. For each transaction, MicroTx generates a new value for the initialization vectors. Each transaction record contains the encrypted metadata information, such as key version and initialization vector value.

3 Prepare

Before you begin installing Transaction Manager for Microservices (MicroTx), set up a transaction store, identity provider, and optionally, a load balancer.

- Transaction store: MicroTx uses a data store for persistence of transaction state. You can use an etcd cluster or an Oracle Database for storing transaction information.
- Identity provider: Use the OpenID Connect JWT tokens to authenticate and authorize user access to MicroTx.
- Load balancer: Optionally, if you set up a load balancer, it must support header-based routing and mTLS.

Topics:

3.1 Download the Installation Bundle

Perform the following steps to download the MicroTx installation bundle to your local system:

1. Visit https://www.oracle.com/database/transaction-manager-for-microservices/.
2. Click **Download** to download either Oracle Transaction Manager for Microservices Enterprise Edition 24.2 or Oracle Transaction Manager for Microservices Free 24.2. You are redirected to Oracle Software Delivery Cloud.
3. Download the MicroTx installation bundle using the Oracle Download Manager or click the ZIP file to download it.
4. Create a new directory in your local machine.

5. Extract the contents of the ZIP file to the new directory that you have created. Unzip the MicroTx installation bundle.

   ```
   unzip otmm-<version>.zip
   ```

6. Run the following command to view the list of files that are extracted.

   ```
   ls -lR otmm-<version>
   ```

The following folders are available.

- `lib`: This folder contains the MicroTx library files. You must use these library files in your application code to use MicroTx to manage transactions amongst your application microservices.
- `otmm`: This folder contains the MicroTx image and `YAML` files which you can use to install and configure MicroTx. The image in this folder is the same image that is available in the Oracle Container Registry. Oracle recommends that you use the MicroTx image from the Oracle Container Registry. For steps to download, see Download the MicroTx image from Oracle Container Registry.
- `console`: This folder contains the image file for the MicroTx console.

3.2 Download the MicroTx image from Oracle Container Registry

The MicroTx image is based on Linux x86-64.

1. Go to Oracle Container Registry.
2. In the search box, type **Transaction Manager for Microservices**, and then click **otmm** in the Search Results page.
3. In the **Tags** section of the page, you can view the available versions of MicroTx. To download the latest version, run one of the following commands:
   - To download the latest version, run the following command:

     ```
     docker pull container-registry.oracle.com/database/otmm:latest
     ```

   - To download a specific version, for example version 24.2, run the following command:

     ```
     docker pull container-registry.oracle.com/database/otmm:24.2
     ```

3.3 Set Up Oracle Identity Providers

You can use Oracle Identity Cloud Service (IDCS) or Oracle IAM as an identity provider to manage access to your application.

If you want to use Keycloak or Microsoft AD as the identity provider, refer to their product documentation for information about setting up the identity provider and creating an access token.

Oracle Cloud Infrastructure previously used Oracle IDCS as the identity provider. Now, Oracle Cloud Infrastructure uses Oracle IAM as the identity provider.

To identify if your Oracle Cloud Infrastructure tenancy uses Oracle IDCS or Oracle IAM:

1. Log in to the Oracle Cloud Infrastructure console.

2. Open the navigation menu and click **Identity & Security**.
   - Under **Identity**, if you see **Users and Groups**, your tenancy has not been migrated to Oracle IAM. Your tenancy uses Oracle IDCS.
   - Under **Identity**, if you see **Domains**, your tenancy has been migrated to Oracle IAM.

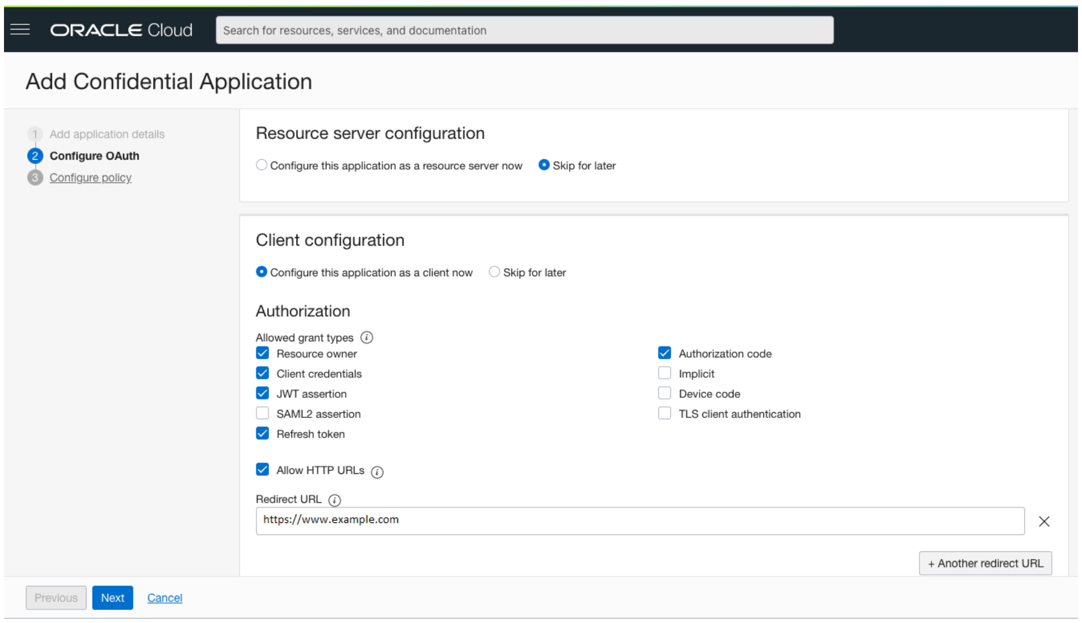
Based on whether your tenancy uses Oracle IDCS or Oracle IAM, you can use the relevant information to create a confidential application and activate it.
Topics:

- Use Oracle IAM as Identity Provider
  You can use Oracle IAM as identity provider to manage access to your application.
- Use Oracle IDCS as Identity Provider
  You can use Oracle IDCS as identity provider to manage access to your application.
- Run the Discovery URL
  After setting up the identity provider, run the Discovery URL in any browser to note down the values that you must provide in the `values.yaml` file for authentication purposes.
- Create an Access Token
  This topic provides details to create an access token when you use Oracle IDCS or Oracle IAM as the identity provider.

3.3.1 Use Oracle IAM as Identity Provider

You can use Oracle IAM as identity provider to manage access to your application.

1. In the Oracle Cloud Infrastructure console, add your application as a confidential application. See Adding a Confidential Application in *Oracle Cloud Infrastructure documentation*.



Add Confidential Application

1 Add application details
2 Configure OAuth
3 Configure policy

Resource server configuration

Configure this application as a resource server now [✓ Skip for later]

Client configuration

[✓ Configure this application as a client now] Skip for later

Authorization

Allowed grant types

✓ Resource owner	✓ Authorization code
✓ Client credentials	Implicit
✓ JWT assertion	Device code
SAML2 assertion	TLS client authentication
✓ Refresh token	

✓ Allow HTTP URLs

Redirect URL: https://www.example.com

While adding a confidential application, perform the following tasks:

a. On the **Configure OAuth** pane, under **Resource server configuration**, click **Skip for later**.

b. On the **Configure OAuth** pane, click **Configure this application as a client now**, and then select the following options:
   - **Resource owner**

- **Client credentials**
- **JWT assertion**
- **Refresh token**
- **Authorization code**
- **Allow HTTP URLs**: Optional. Select this option only if you want to add a redirect URL without HTTPS. If you don't select this option, only HTTPS URLs are supported.
- **Add Redirect URL**: Enter the application URL where the user is redirected after authentication.

c. Skip web tier policy configuration.

The application is created.

2. Click **Activate** to activate the application.
3. Under **General Information**, note down the values for **Client ID** and **Client secret**.
4. Click **Users**, and then assign users to the application. See Assigning Users to Custom Applications in *Oracle Cloud Infrastructure documentation*.
5. Open the navigation menu and click **Identity & Security**. Under **Identity**, click **Domains**. Select the identity domain you want to work in.

   The **Domain information** tab of the identity domain is displayed.
6. From this tab, copy the **Domain URL**. For example, `https://idcs-a83e4de370ea4db1b8c703a0b742ce74.identity.oraclecloud.com`. You'll need this information while running the Discovery URL.
7. Enable client access for the signing certificate. By default, access is restricted to only the signed-in users. To access this certificate in Docker, Kubernetes, and Istio, you must enable client access.

   a. Select the identity domain you want to work in and click **Settings** and then **Domain settings**.

   b. Turn on the switch under **Access Signing Certificate** to enable clients to access the tenant signing certificate without logging in to IAM.

   c. Click **Save** to save the default settings.

   d. To check if you can access the certificate without logging in, type the following link in a new browser window.

   ```
   https://<yourtenant>.identity.oraclecloud.com/admin/v1/SigningCert/jwk
   ```

   Where, *<yourtenant>* are the details of your Oracle Cloud Infrastructure tenancy.

   You should be able to open the link without logging in to Oracle Cloud Infrastructure.

3.3.2 Use Oracle IDCS as Identity Provider

You can use Oracle IDCS as identity provider to manage access to your application.

1. In the Oracle Cloud Infrastructure console, add your application as a confidential application. See Adding a Confidential Application in *Administering Oracle Identity Cloud Service*.

   While adding a confidential application, perform the following tasks:

   a. On the **Add Confidential Application** wizard's **Client** page, click **Configure this application as a client now**.
   b. In the **Authorization** section, select the following options:
      - **Resource owner**
      - **Client credentials**
      - **JWT assertion**
      - **Refresh token**
      - **Authorization code**
      - **Redirect URL**: Enter the application URL where the user is redirected after authentication.
   c. Skip the next steps. Use the default selections, and then click **Finish**. The application has been added in a deactivated state.
   d. Record the **Client ID** and **Client Secret** that appear in the **Application Added** dialog box. You will need to provide this information later.
   e. Click **Close**.
      The new application's details page is displayed.
   f. At the top of the page, to the right of the application name, click **Activate** to activate the application.
   g. In the **Activate Application?** dialog box, click **Activate Application**.
2. Click **Users**, and then assign users to the application. See Assign Applications to the User Account in *Administering Oracle Identity Cloud Service*.
3. Enable client access for the signing certificate. By default, access is restricted to only the signed-in users. To allow clients to access the tenant signing certificate and the SAML metadata without logging in to Oracle Identity Cloud Service, perform the following steps.
   a. In the Identity Cloud Service console, expand the **Navigation Drawer**, click **Settings**, and then click **Default Settings**.
   b. Turn on the **Access Signing Certificate** option.
   c. Click **Save** to save the default settings.

3.3.3 Run the Discovery URL

After setting up the identity provider, run the Discovery URL in any browser to note down the values that you must provide in the `values.yaml` file for authentication purposes.

To run the Discovery URL and note down the required identity provider information:

1. Run the Discovery URL in any browser.

   Syntax of Discovery URL

   ```
   https://<tenant-base-url>/.well-known/openid-configuration
   ```

   Example Discovery URL

   ```
   https://idcs-a83e....identity.oraclecloud.com/.well-known/openid-
   configuration
   ```

The example tenant base URL has been truncated with ellipses (...) for readability. When you run this command in your environment, copy the actual value.

A list of values is displayed.

2. Note down the values for the `issuer` and `jwksUri` fields. You will need to provide these values in the `values.yaml` file. For example:

```
issuer: "https://identity.oraclecloud.com"
jwksUri: "https://idcs-a83e....identity.oraclecloud.com:443/admin/v1/
SigningCert/jwk"
```

3. Note down the value of the `token_endpoint` field. You will provide this information in the `values.yaml` file as value for the `tmmConfiguration.identityProvider.identityProviderUrl` property.

```
token_endpoint: "https://idcs-a83e.....identity.oraclecloud.com/oauth2/v1/
token"
```

3.3.4 Create an Access Token

This topic provides details to create an access token when you use Oracle IDCS or Oracle IAM as the identity provider.

If you want to use Keycloak or Microsoft AD as the identity provider, refer to their product documentation for information about setting up the identity provider and creating an access token.

API calls to the service require a valid access token. Create an access token which you can specify in subsequent API calls to the service. In addition to the access token, you can also specify the refresh token in subsequent API calls to the service. MicroTx uses the refresh token to refresh an expired access token.

Before you begin, ensure that you have set up your identity provider and noted down the values for client ID, client secret, and the domain URL.

1. Create an access token. See Example Authorization Flow in *REST API for Oracle Identity Cloud Service*.
2. Store the access token and refresh tokens in environment variables, as shown in the following example for a Linux host.

```
export TOKEN="eyJ4Lm..."
export REFRESH_TOKEN="ey5Gkr..."
```

3. Store the authentication cookie in an environment variable, as shown in the following example for a Linux host.

```
export OTMM_COOKIE="eyJh...x_THw"
```

The example value has been truncated with ellipses (...) for readability.

After you obtain the OAuth 2.0 tokens, use the tokens in the `authorization` and `refresh-token` headers while making subsequent API calls to the service.

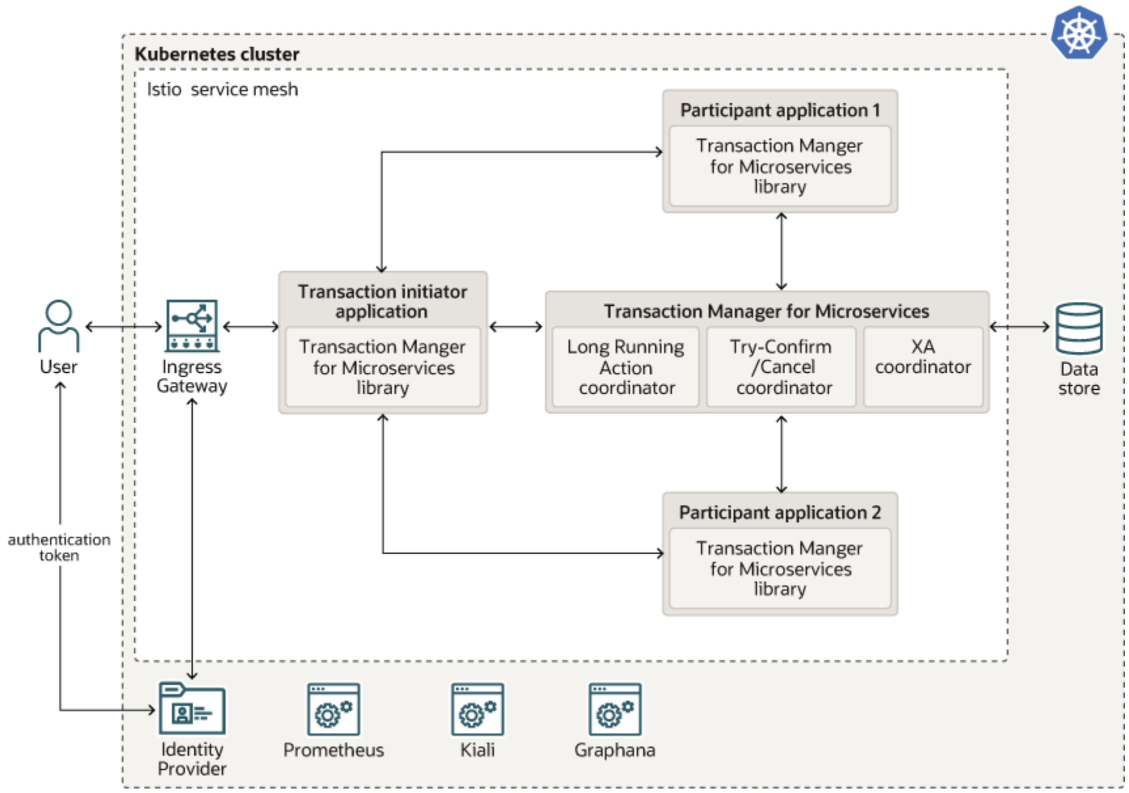
3.4 Prepare a Kubernetes Cluster

In Kubernetes, you can install MicroTx 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. This section provides instructions to install MicroTx on a Kubernetes cluster with and without using the Istio service mesh.

You can create a similar configuration to install MicroTx 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 is installed in a Kubernetes cluster within an Istio service mesh along with other microservices.


Kubernetes cluster
Istio service mesh
Participant application 1
Transaction Manger for Microservices library
Transaction initiator application
Transaction Manger for Microservices library
Transaction Manager for Microservices
Long Running Action coordinator
Try-Confirm /Cancel coordinator
XA coordinator
Data store
User
Ingress Gateway
authentication token
Participant application 2
Transaction Manger for Microservices library
Identity Provider
Prometheus
Kiali
Graphana

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.

Topics:

- Considerations for Deployment on Kubernetes
  Consider the following factors while deploying MicroTx on Kubernetes.
- Create a Kubernetes Cluster
  Create a Kubernetes cluster or use an existing one. You will install MicroTx onto this cluster.

- Install the Required Software for Kubernetes
  Before installing MicroTx in Kubernetes, you must install and configure the required software on your local machine.
- Install the Required Software for OpenShift
  Before installing MicroTx in OpenShift, you must install and configure the required software on your local machine.
- Install and Configure Istio
  Install Istio, 1.12.1 or later versions, onto the Kubernetes cluster with the default Istio profile.
- Create a Kubernetes Secret with SSL Details for Istio
  To enable access to Istio using the HTTPS protocol, you must create a Kubernetes secret that contains details of an SSL key and certificate. MicroTx uses this information to access Istio using HTTPS.
- Create a Kubernetes Secret to Access Docker Registry
  When you install the application using Helm, use a Kubernetes secret to provide the authentication details to pull an image from the remote repository.
- Authenticate and Authorize
  Authentication ensures that only authorized individuals get access to the system and data. Authorization provides access control to system privileges and data. This builds on authentication to ensure that individuals get appropriate access.

3.4.1 Considerations for Deployment on Kubernetes

Consider the following factors while deploying MicroTx on Kubernetes.

The installation bundle provides Helm charts and this document provides details for a sample deployment of MicroTx in a Kubernetes cluster with Istio service mesh. If you are using another service mesh in a Kubernetes cluster, create your own Helm charts.

Supported Kubernetes Platforms

Deploy MicroTx in a Kubernetes cluster that is running in your data center or your cloud environment. MicroTx is tested with Kubernetes 1.21.x or compatible versions on the following platforms:

- Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE). See Creating a Kubernetes Cluster in *Oracle Cloud Infrastructure documentation*.
- Minikube
- Oracle Linux Container Native Environment

Deployment Across Multiple Kubernetes Clusters

You can deploy your application microservices and MicroTx within a single Istio service mesh in a single Kubernetes cluster.

When your application microservices are distributed across multiple Kubernetes cluster, or if you want MicroTx to communicate with Oracle Database or Tuxedo, then you can deploy MicroTx in a separate Kubernetes cluster. In such a scenario, each Kubernetes cluster will contain an Istio service mesh. You will have to configure ingress and egress gateways to enable communication between multiple Istio services meshes.

3.4.2 Create a Kubernetes Cluster

Create a Kubernetes cluster or use an existing one. You will install MicroTx onto this cluster.

Before you begin, you must plan the environment in the following way:

- Decide if you require a single-node or a multinode Kubernetes cluster to host MicroTx. Oracle recommends that you create at least a single-node cluster in development environments and at least a three-node cluster in production environments.
- Identify the different components in your environment. If your microservices are running in a Kubernetes cluster, you can install MicroTx in the same cluster or a different cluster. If your microservices are distributed across multiple Kubernetes clusters or if you want MicroTx to communicate with components such as Oracle Database or Tuxedo, which are not part of any Kubernetes cluster, create a Kubernetes cluster to host MicroTx.

3.4.3 Install the Required Software for Kubernetes

Before installing MicroTx in Kubernetes, you must install and configure the required software on your local machine.

Skip this step if you are using OpenShift.

Perform the following steps to install the required software and configure the environment in your local machine:

1. Install and configure Kubernetes command-line interface (Kubectl), 1.21.x or later versions, to create and manage your deployments in your Kubernetes cluster. See https://kubernetes.io/docs/tasks/tools/.
2. Install the latest version of Helm 3.x on your local machine. For more information, see https://helm.sh/docs/intro/install/.

   Use Helm to make deployments easier as you can run a single command to install applications and resources into Kubernetes clusters. Helm interacts with the Kubernetes API server to install, upgrade, query, and remove Kubernetes resources.
3. Create a namespace. The following command creates a namespace with the name `otmm`.

   Sample Command

   ```
   kubectl create ns otmm
   ```

   Sample Response

   ```
   namespace/otmm created
   ```

   Note down the name of the namespace. Later, you will deploy MicroTx in this namespace.

3.4.4 Install the Required Software for OpenShift

Before installing MicroTx in OpenShift, you must install and configure the required software on your local machine.

Skip this step if you are using Kubernetes.

Perform the following steps to install the required software and configure the environment in your local machine:

1. Install and configure OpenShift command-line interface, 4.14.x or later versions, to create and manage your deployments in OpenShift cluster. See https://docs.openshift.com/container-platform/4.15/cli_reference/index.html.
2. Install the latest version of Helm 3.x on your local machine. For more information, see https://helm.sh/docs/intro/install/.

   Use Helm to make deployments easier as you can run a single command to install applications and resources into Kubernetes clusters. Helm interacts with the Kubernetes API server to install, upgrade, query, and remove Kubernetes resources.
3. Create a project. The following command creates a project with the name `otmm`.

   Sample Command

   ```
   oc new-project otmm
   ```

   Note down the name of the project. Later, you will deploy MicroTx in this project.

3.4.5 Install and Configure Istio

Install Istio, 1.12.1 or later versions, onto the Kubernetes cluster with the default Istio profile.

Skip this section if you don't want to install MicroTx in an Istio service mesh.

Oracle recommends using the default Istio profile for production environments. Additionally, create logs that you can use for audits and enable distributed tracing to monitor and troubleshoot microservices-based distributed systems, such as monitoring distributed transactions, analyzing the root cause, analyze service dependency, and optimize performance or latency. For more information, see https://istio.io/latest/docs/setup/additional-setup/config-profiles/.

Perform the following steps to install and configure Istio on your local machine:

1. Run the following command to download Istio.

   ```
   curl -sL https://istio.io/downloadIstioctl | sh -
   ```

2. Move to the Istio package directory. For example, if the package is `istio-1.12.1`:

   ```
   cd istio-1.12.1
   ```

3. Add the `istioctl` client tool which is located in the `bin` folder to the `PATH` for your workstation. The following example specifies the a sample value. Provide the path based on your environment.

   ```
   export PATH=$PWD/bin:$PATH
   ```

4. Run prerequisite checks to validate if the cluster meets Istio install requirements.

   ```
   istioctl x precheck
   ```

   The following message is displayed. You can proceed with the next step and install Istio if there are no issues.

   ```
   No issues found when checking the cluster. Istio is safe to install or
   upgrade!
   ```

5. Run the following commands to install and configure Istio in Kubernetes. Skip this step if you are using OpenShift.
   a. Run the following command to install Istio on Kubernetes with the default Istio profile.

      ```
      istioctl install --set meshConfig.accessLogFile=/dev/stdout \
          --set meshConfig.accessLogEncoding=JSON \
          --set meshConfig.enableTracing=true \
          --set meshConfig.defaultConfig.tracing.sampling=100.0
      ```

      Only in Oracle Linux 8 environments, run the following command to pass an additional flag, `--set components.cni.enabled=true`. For example:

      ```
      istioctl install --set meshConfig.accessLogFile=/dev/stdout \
          --set meshConfig.accessLogEncoding=JSON \
          --set meshConfig.enableTracing=true \
          --set meshConfig.defaultConfig.tracing.sampling=100.0 \
          --set components.cni.enabled=true
      ```

   b. Label the namespace that you have created with `istio-injection=enabled` to put automatic sidecar injection into effect. The following command labels the `otmm` namespace:
      Sample Command

      ```
      kubectl label namespace otmm istio-injection=enabled
      ```

      Sample Response

      ```
      namespace/otmm labeled
      ```

6. Run the following commands to install and configure Istio in OpenShift. Skip this step for Kubernetes.
   a. Run the following command to install Istio on OpenShift with the default Istio profile.

      ```
      istioctl install --set profile=openshift \
          --set meshConfig.accessLogFile=/dev/stdout \
          --set meshConfig.accessLogEncoding=JSON \
          --set meshConfig.enableTracing=true \
          --set meshConfig.defaultConfig.tracing.sampling=100.0 \
          --set components.cni.enabled=true
      ```

   b. After Istio is installed, expose an OpenShift route for the Istio ingress gateway.

      ```
      oc -n istio-system expose svc/istio-ingressgateway --port=http2
      ```

   c. Label the project that you have created with `istio-injection=enabled` to put automatic sidecar injection into effect. The following command labels the `otmm` project in OpenShift:
      Sample Command

      ```
      oc label project otmm istio-injection=enabled
      ```

3.4.6 Create a Kubernetes Secret with SSL Details for Istio

To enable access to Istio using the HTTPS protocol, you must create a Kubernetes secret that contains details of an SSL key and certificate. MicroTx uses this information to access Istio using HTTPS.

Before you begin, ensure that you have installed Istio in the Kubernetes cluster. See Install the Required Software for Kubernetes.

To create Kubernetes secret with the details of the SSL certificates:

1. Identify the location of the SSL certificates that you want to use.
2. Import the SSL certificates to the Kubernetes namespace where you have installed Istio. Run the following command to create a Kubernetes secret with details of the SSL certificate.

   Syntax

   ```
   kubectl create secret tls tls-credential --key=ssl-key-file-path --
   cert=ssl-cert-file-path -n istio-system
   ```

   Example

   ```
   kubectl create secret tls tls-credential --key=~/certificates/
   example.dev.key --cert=~/certificates/example.dev.crt -n istio-system
   ```

   Where,

   - `tls` is the type of the secret.
   - *`tls-credential`* is the name of the Kubernetes secret that you want to create.
   - *`ssl-key-file-path`* is the location of the SSL certificate key file.
   - *`ssl-cert-file-path`* is the location of the SSL certificate file.
   - *`istio-system`* is the namespace in which you have installed Istio. The default namespace is `istio-system`. If you have installed Istio in another namespace, run the `kubectl get ns` command to find all the namespaces in the cluster.

Note down the name of the Kubernetes secret as you will need to provide this detail in the `values.yaml` file to install MicroTx.

3.4.7 Create a Kubernetes Secret to Access Docker Registry

When you install the application using Helm, use a Kubernetes secret to provide the authentication details to pull an image from the remote repository.

The Kubernetes Secret contains all the login details you provide if you were manually logging in to the remote Docker registry using the `docker login` command, including your credentials.

1. Create a secret by providing the credentials on the command-line by using the following command.

   ```
   kubectl create secret docker-registry NAME --docker-server=SERVER --docker-
   username=USERNAME --docker-password=PASSWORD --docker-email=EMAIL --
   namespace=NAMESPACE
   ```

Where,

- `NAME`: Name of the Kubernetes secret that you want to create. Note down this name as you will use this name later in the manifest file to refer to the secret.
- `SERVER`: Name of your private Docker registry. The format varies based on your Kubernetes platform. For example, the format of the user name in Oracle Cloud Infrastructure environment is `<region-key>.ocir.io`.
- `USERNAME`: User name to access the remote Docker registry. The format varies based on your Kubernetes platform. For example, the format of the user name in Oracle Cloud Infrastructure environment is `<tenancy-namespace>/<oci-username>`.
- `PASSWORD`: Password to access the remote Docker registry.
- `EMAIL`: Email ID for your Docker registry.
- `NAMESPACE`: Namespace where you want to deploy MicroTx.

Example

Use the following command to create a Kubernetes secret with the name *regcred* in the `otmm` namespace.

```
kubectl create secret docker-registry regcred --docker-server=iad.ocir.io
--docker-username=mytenancy/myuser --docker-password=pwd --docker-
email=myuser@example.com --namespace=otmm
```

2. Note down the name of the secret that you have created. You will need to provide this value later.
3. Close the terminal.

   When you type secrets at the command line, the command line may store the secrets in your shell history unprotected. The secrets might also be visible to other users on your PC during the time that kubectl is running. To overcome this issue, you can close the terminal after creating the secret.

You can also create a secret based on existing credentials. See https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/#registry-secret-existing-credentials.

3.4.8 Authenticate and Authorize

Authentication ensures that only authorized individuals get access to the system and data. Authorization provides access control to system privileges and data. This builds on authentication to ensure that individuals get appropriate access.

- Generate a Kubernetes Secret for an Encryption Key
  To support asynchronous calls, MicroTx stores the authorization and refresh tokens. To store the tokens, you have to encrypt it as you can't store the token directly. To encrypt the tokens, create encryption keys.
- Create a Key Pair for Transaction Token
  The application supports including a MicroTx signed transaction token which is unique to each MicroTx transaction.

3.4.8.1 Generate a Kubernetes Secret for an Encryption Key

To support asynchronous calls, MicroTx stores the authorization and refresh tokens. To store the tokens, you have to encrypt it as you can't store the token directly. To encrypt the tokens, create encryption keys.

MicroTx encrypts the tokens using the encryption keys that you provide. When there is an asynchronous call from MicroTx to participant services, MicroTx fetches the encrypted token, decrypts it, and then attaches the token to the authorization header.

You must generate an encryption key, and then add the key to a Docker secret if you have enabled the `authTokenPropagationEnabled` property under `authorization`. The encryption key that you generate must have the following attributes.

- Symmetric algorithm: AES-256
- Cipher mode: AES in GCM mode
- Key length: 32 bytes
- Length of initialization vectors: 96 bits

MicroTx encrypts the access and refresh tokens, and then uses it later while making calls to participant services. For each transaction, MicroTx generates a new value for the initialization vectors. Each transaction record contains the encrypted metadata information, such as key version and initialization vector value.

1. Run the following command to generate an encryption key with a key length of 32 bytes.

   ```
   openssl rand -hex 16
   ```

   Note down the value that is generated. For example, `e9f0adab17c0180425147166c2ff1cd3`.

2. Create a Kubernetes secret while using the encrypted key that you have generated as the value. You must create this secret in the namespace where you want to install MicroTx.

   The following sample command creates a Kubernetes secret with the name `encryption-secret-key1` in the `otmm` namespace.

   ```
   kubectl create secret generic encryption-secret-key1 \ --from-
   literal=secret='e9f0adab17c0180425147166c2ff1cd3' -n otmm
   ```

3. Note down the name of the Kubernetes secret and its version. You will provide these values to for the `secretKeyName` and `version` fields in the `values.yaml` file.

   The following code snippet provides sample values for the `encryption` field in the `values.yaml` file. The sample values in this example are based on the values used in the sample commands in this topic.

   ```
   encryption:
     encryptionSecretKeyVersion: "1"
     encryptionSecretKeys:
         - secretKeyName: "encryption-secret-key0"
           version: "0"
         - secretKeyName: " encryption-secret-key1"
           version: "1"
   ```

3.4.8.2 Create a Key Pair for Transaction Token

The application supports including a MicroTx signed transaction token which is unique to each MicroTx transaction.

When you set `transactionTokenEnabled` to true, MicroTx creates a new token called `tmm-tx-token`, which is a signed transaction token. When transaction initiator begins a request, the MicroTx responds with the `tmm-tx-token`. To secure calls from the participant services to MicroTx, the MicroTx library passes `tmm-tx-token` in the request header. You don't have to create the `tmm-tx-token` transaction token or pass it in the request header. The MicroTx library creates this token based on the private-public key pair that you provide.

The transaction token that you generate must have the following attributes:

- Asymmetric algorithm: RSA 3072
- Key length: 3072 bits
- Hash algorithm: SHA256

Before you begin, ensure that you have installed OpenSSL.

1. Create RSA private key with key length as 3072 bits by using the following command:

   ```
   openssl genrsa -aes256 -out private.pem 3072
   ```

2. Enter a pass phrase at the command prompt, and then press enter. Remember the pass phrase as you will have to provide it later.

   A new file called `private.pem` is created in the current working folder. This file contains the RSA private key value.

3. Create a RSA public key for the private key that you have generated. Use the following command:

   The following command creates a new file called `public.pem` in the current working folder. This file contains the RSA public key value.

   ```
   openssl rsa -in private.pem -outform PEM -pubout -out public.pem
   ```

4. Run the following command to base64 encode the `private.pem` file.

   Example command

   ```
   base64 private.pem
   ```

   The base64-encoded value of the `private.pem` file is returned.

   Example response

   ```
   LS0tLS...LS0tLQo=
   ```

   The example response has been truncated with ellipses (...) for readability.

   Note down the base64-encoded value of the `private.pem` file.

5. Create a Kubernetes secret with the base64-encoded value of the `private.pem` file.

The following command creates a Kubernetes secret with the name `TMMPRIVKEY1` in the `otmm` namespace, where you want to install MicroTx.

```
kubectl create secret generic TMMPRIVKEY1 \ --from-
literal=secret='LS0tLS...LS0tLQo=' -n otmm
```

Note down the name of the Kubernetes secret. You will need to provide this value later in the `values.yaml` file.

6. Run the following command to base64 encode the `public.pem` file.

   Example command

   ```
   base64 public.pem
   ```

   The base64-encoded value of the `public.pem` file is returned.

   Example response

   ```
   LS0tLS...LS0tCg==
   ```

   The example response has been truncated with ellipses (...) for readability.

   Note down the base64-encoded value of the `public.pem` file.

7. Create a Kubernetes secret with the base64-encoded value of the `public.pem` file.

   The following command creates a Kubernetes secret with the name `TMMPUBKEY1` in the `otmm` namespace.

   ```
   kubectl create secret generic TMMPUBKEY1 \ --from-
   literal=secret='LS0tLS...LS0tCg==' -n otmm
   ```

   Note down the name of the Kubernetes secret. You will need to provide this value later in the `values.yaml` file.

8. Create Kubernetes secret with the value as private key pass phrase that you had provided in step 2.

   The following command creates a Kubernetes secret with the name `TMMPRIVKEYPASSWD1` and key pass phrase as *`<pph...>`* in the `otmm` namespace.

   ```
   kubectl create secret generic TMMPRIVKEYPASSWD1 \ --from-
   literal=secret='<pph...>' -n otmm
   ```

   Where, *`<pph...>`* is the private key pass phrase. Replace this with a value specific to your environment.

   Note:

   Do not base64-encode the key pass phrase, as you must enter the key pass phrase in plain-text format.

Note down the name of the Kubernetes secret. You will need to provide this value later in the `values.yaml` file.

3.5 Set Up Access to MicroTx Web Console

Complete all the tasks in this section to ensure that users can access the MicroTx web console. Skip this section if you don't want to set up the web console.

Topics:

- Specify the Admin Role in YAML file
  Create and assign the administrator role to users in your identity provider. After creating the admin role, update the YAML file with the name of the admin role.
- Create a Secret with Identity Provider Client Credentials
  To provide users the capability to log in to the MicroTx console, you must set up an identity provider. Create a Kubernetes secret to provide the client credentials of the identity provider to the MicroTx.
- Create a Secret with Cookie Encryption Password for Kubernetes
  In Kubernetes environment, the MicroTx console requires the session cookie to be encrypted. You must provide a cookie encryption password through a Kubernetes secret.
- Create a Secret with Cookie Encryption Password for OpenShift
  The MicroTx console requires the session cookie to be encrypted, so you must provide the cookie encryption password through a Kubernetes secret in OpenShift environments.
- Deploy Kubernetes Metrics Server
  To collect metrics about the health of the MicroTx coordinator and console, deploy the Metrics Server.

3.5.1 Specify the Admin Role in YAML file

Create and assign the administrator role to users in your identity provider. After creating the admin role, update the YAML file with the name of the admin role.

Administrators can use the MicroTx console and MicroTx REST APIs to view and manage *all* transactions. Other users can manage and view only the distributed transactions that they have initiated; they cannot view the transactions initiated by other users.

1. Create an administrator role in your identity provider.

   Note down the name of the role. You will need to provide this as the value of the `tmmConfiguration.identityProvider.adminUserRoles` property in the `values.yaml` file.
2. Assign the administrator role to users.

   For information about creating administrator role and assigning it to a user, refer to your identity provider documentation.
3. Note down the path to the administrator role from the JWT access token. You will provide this value for the `adminUserRolesPath` property in the `values.yaml` file.

   - The following sample code snippet provides the role paths in the JWT access token for Oracle IDCS. For JWT token in Oracle IDCS, the roles are present under `userAppRoles`. Based on the sample provided below the value for the `adminUserRolesPath` property, in the `values.yaml` file, is `userAppRoles`.

```
"userAppRoles": [
    "Identity Domain Administrator"
  ]
```

- The following sample provides the role paths in the access token for Keycloak. Based on the sample provided below the value for the `adminUserRolesPath` property, in the `values.yaml` file, is `realm_access, roles`.

```
# Admin role path in the JWT token from root.
    # Ex: For keycloak payload, roles are present under realm_access
    # {
    #   "realm_access": {
    #     "roles": [
    #       "admin"
    #     ]
    #   }
    # }
```

3.5.2 Create a Secret with Identity Provider Client Credentials

To provide users the capability to log in to the MicroTx console, you must set up an identity provider. Create a Kubernetes secret to provide the client credentials of the identity provider to the MicroTx.

Before you begin, ensure that you have set up your identity provider and noted down the values for client ID and client secret.

To create a Kubernetes secret with the identity provider client credentials:

1. Launch a terminal and enter the following commands to base64 encode the client ID and client secret.

   ```
   echo -n "clientid" | base64 -w 0
   echo -n "clientSecret" | base64 -w 0
   ```

   Replace `clientid` and `clientSecret` with the values in your environment.

   Note:

   For Linux, add `-w 0` to the command to remove line breaks.

   The base64 encoded value of the client ID and client secret is returned. Note down these values as you will need it later.

2. Paste the following code in any text editor.

   ```
   apiVersion: v1
       kind: Secret
       metadata:
   ```

```
  name: console-identity-client-secret
type: Opaque
data:
  clientId: base64_encoded_clientId
  clientPassword: base64_encoded_clientSecret
```

Where,

- *`console-identity-client-secret`* is the name of the Kubernetes secret that you want to create. Note down this name as you will have to provide it later in the `values.yaml` file.
- *`base64_encoded_clientId`* and *`base64_encoded_clientSecret`* are the base64 encoded values of the client ID and client secret that you have generated in the previous step.

Replace these with values specific to your environment.

3. Save the file as a YAML file. For example, `consoleSecret.yaml`.
4. Run the following command to create a Kubernetes secret in the namespace where you want to install MicroTx.

   Command syntax

   ```
   kubectl apply -f <filename> -n <namespace>
   ```

   The following sample command creates a Kubernetes secret with the name `console-identity-client-secret` in the `otmm` namespace with the details that you have provided in the `consoleSecret.yaml` file.

   ```
   kubectl apply -f consoleSecret.yaml -n otmm
   ```

Note down the name of the secret, `console-identity-client-secret`. You'll provide this name as the value for the `tmmConfiguration.identityProvider.clientSecretName` property in the `values.yaml` file.

3.5.3 Create a Secret with Cookie Encryption Password for Kubernetes

In Kubernetes environment, the MicroTx console requires the session cookie to be encrypted. You must provide a cookie encryption password through a Kubernetes secret.

To encrypt the MicroTx console session cookie, create a Kubernetes secret with the cookie encryption password:

1. Launch a terminal and enter the following commands to base64 encode the cookie encryption password.

   ```
   echo -n "cookie-encryption-password" | base64 -w 0
   ```

   Replace `cookie-encryption-password` with a password of your choice.

   > **Note:**
   >
   > For Linux, add `-w 0` to the command to remove line breaks.

The base64 encoded value of the password is returned. Note down this value as you will need it later.

2. Paste the following code in any text editor.

```
apiVersion: v1
    kind: Secret
    metadata:
      name: console-cookie-encryption-password-secret
    type: Opaque
    data:
      secret: base64_encoded_cookieEncryptionPassword
```

   Where,

   - `console-cookie-encryption-password-secret` is the name of the Kubernetes secret that you want to create. Note down this name as you will have to provide it later in the `values.yaml` file.
   - `base64_encoded_cookieEncryptionPassword` is the base64 encoded value of the password that you have generated in the previous step.

3. Save the file as a YAML file. For example, `cookieEncryptionPassword.yaml`.
4. Run the following command to create a Kubernetes secret in the namespace where you want to install MicroTx.

   Command syntax

```
kubectl apply -f <filename> -n <namespace>
```

   The following sample command creates a Kubernetes secret with the name `console-cookie-encryption-password-secret` in the `otmm` namespace with the details that you have provided in the `cookieEncryptionPassword.yaml` file.

```
kubectl apply -f cookieEncryptionPassword.yaml -n otmm
```

Note down the name of the secret, `console-cookie-encryption-password-secret`. You'll provide this name as the value for the `tmmConfiguration.tmmConsoleConfiguration.cookieEncryptionPasswordSecretName` property in the `values.yaml` file.

3.5.4 Create a Secret with Cookie Encryption Password for OpenShift

The MicroTx console requires the session cookie to be encrypted, so you must provide the cookie encryption password through a Kubernetes secret in OpenShift environments.

To encrypt the MicroTx console session cookie, create a Kubernetes secret with the cookie encryption password:

1. In the OpenShift environment, switch to project where you want to deploy MicroTx.

```
oc project otmm
```

   Where, `otmm` is the name of the project that you have created. See Install the Required Software for OpenShift.

2. Launch a terminal and enter the following commands to base64 encode the cookie encryption password.

   ```
   echo -n "cookie-encryption-password" | base64 -w 0
   ```

   Replace `cookie-encryption-password` with a password of your choice.

   > **Note:**
   >
   > For Linux, add `-w 0` to the command to remove line breaks.

   The base64 encoded value of the password is returned. Note down this value as you will need it later.

3. Paste the following code in any text editor.

   ```
   apiVersion: v1
       kind: Secret
       metadata:
         name: console-cookie-encryption-password-secret
       type: Opaque
       data:
         secret: base64_encoded_cookieEncryptionPassword
   ```

   Where,

   - *`console-cookie-encryption-password-secret`* is the name of the Kubernetes secret that you want to create. Note down this name as you will have to provide it later in the `values.yaml` file.
   - *`base64_encoded_cookieEncryptionPassword`* is the base64 encoded value of the password that you have generated in the previous step.

4. Save the file as a YAML file. For example, `cookieEncryptionPassword.yaml` in the `tmm-istio-openshift/resources` folder.

5. Run the following command to create a Kubernetes secret in the project where you want to install MicroTx.

   Command syntax

   ```
   oc apply -f <filename>
   ```

   The following sample command creates a Kubernetes secret with the name `console-cookie-encryption-password-secret` with the details that you have provided in the `cookieEncryptionPassword.yaml` file.

   ```
   oc apply -f tmm-istio-openshift/resources/cookie-encryption-secret.yaml
   ```

Note down the name of the secret, `console-cookie-encryption-password-secret`. You'll provide this name as the value for the `tmmConfiguration.tmmConsoleConfiguration.cookieEncryptionPasswordSecretName` property in the `values.yaml` file.

3.5.5 Deploy Kubernetes Metrics Server

To collect metrics about the health of the MicroTx coordinator and console, deploy the Metrics Server.

- Run the following command to deploy the Kubernetes Metrics Server:

  ```
  kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/
  releases/latest/download/components.yaml
  ```

For more information about the Metrics Server, see https://github.com/kubernetes-sigs/metrics-server/tree/master?tab=readme-ov-file#kubernetes-metrics-server.

3.6 Set Up etcd as Data Store

Create a data store for MicroTx to store the transaction logs. You can use either etcd or Oracle Database as the data store.

Skip this section if you want to set up Oracle Database as data store. See Set Up Oracle Database as Data Store.

Before installing MicroTx, you must install and configure a data store. Ensure that you set up the required networking rules to allow communication between the transaction coordinator and the data store.

Ensure that you have the required permissions to create tables in the database. When you install MicroTx, the service creates the required tables in the database. So MicroTx requires certain details about the database.

Topics:

- Generate RSA Certificates for etcd
  You must provide etcd credentials and etcd endpoints in the `YAML` file for the transaction coordinator. MicroTx uses this information to establish a connection to the database after the service is installed.
- Create a Kubernetes Secret for etcd
  You must provide etcd credentials and etcd endpoints in the `values.yaml` file. MicroTx uses this information to establish a connection to etcd after the service is installed.

3.6.1 Generate RSA Certificates for etcd

You must provide etcd credentials and etcd endpoints in the `YAML` file for the transaction coordinator. MicroTx uses this information to establish a connection to the database after the service is installed.

Skip this step if you are not using etcd as the transaction store.

Before you begin, complete the following tasks:

- Install CFSSL tool. See https://github.com/cloudflare/cfssl. This topic provides sample commands to create certificates using the CFSSL tool. You can use this tool or any other tool of your choice to generate certificates.

- Install and configure the etcd database. For information to create an etcd data store, see https://etcd.io/docs/.
- Enable TLS on etcd for additional security and provide the certificate details in the `YAML` file for the transaction coordinator.

To create certificates and identify the etcd endpoints:

1. Create a directory.

   The following sample code creates a directory named, `cfssl`.

   ```
   mkdir cfssl
   cd cfssl
   ```

   Note the path of this directory as you will create all the certificates inside it.
2. Run the following command to identify the external IP address of the etcd database server.

   Run the following command only if you want to install MicroTx in a Kubernetes cluster.

   ```
   kubectl get svc
   ```

   Sample output

   ```
   NAME           TYPE           CLUSTER-IP      EXTERNAL-IP
   PORT(S)              AGE

   etcd           ClusterIP      None            <none>           4002/
   TCP,4003/TCP   5h8m

   etcd-client    LoadBalancer   192.0.2.83     198.51.100.1
   4002:32135/TCP       5h8m
   ```
3. Note down the external IP address.

   You will provide this value to generate the server certificate and as the etcd endpoints in the `YAML` file for the transaction coordinator.
4. Run the following command to initialize certificate authority.

   ```
   echo '{"CN":"CA","key":{"algo":"rsa","size":2048}}' | cfssl gencert -
   initca - | cfssljson -bare ca -
   ```

   This command creates three files in the current working directory: `ca-key.pem`, `ca.csr`, and `ca.pem` files.
5. Run the following command to configure the certificate authority options.

   Sample command

   ```
   echo '{"signing":{"default":{"expiry":"43800h","usages":["signing","key
   encipherment","server auth","client auth"]}}}' > ca-config.json
   ```

   Where, the output is written to the `ca-config.json` file.

   You can modify values for `expiry` and `usages`. For more information about these attributes, refer to the CFSSL documentation.

6. Generate the server certificate.

   a. Run the following command to assign the IP address of the etcd database server to the variable `ADDRESS`. When you run this command in your environment, replace the sample value with a value specific to your environment.

      ```
      export ADDRESS=192.0.2.82
      ```

   b. Run the following command to assign the name of the etcd database server to the variable `NAME`. This is the server Common Name (CN) that is required to generate the server certificate. When you run this command in your environment, replace the sample value with a value specific to your environment.

      ```
      export NAME=server
      ```

   c. Run the following command to generate the server certificate.

      ```
      echo '{"CN":"'$NAME'","hosts":[""],"key":{"algo":"rsa","size":2048}}' |
      cfssl gencert -config=ca-config.json -ca=ca.pem -ca-key=ca-key.pem -
      hostname="$ADDRESS" - | cfssljson -bare $NAME
      ```

   This command creates three files in the current working directory: `server-key.pem`, `server.csr`, and `server.pem` files.

7. Add permissions to the server certificate. Perform this step only if you want to install MicroTx in Docker Swarm. Skip this step if you want to install MicroTx in a Kubernetes cluster.

   ```
   sudo chmod 644 server-key.pem
   sudo chmod 644 server.pem
   ```

8. Generate the client certificate. While generating the client certificate, you don't need to specify an IP address for the client certificate host.

   a. Run the following command to assign a name to the variable `NAME`. This is the server Common Name (CN) that is required to generate the client certificate. You can provide any value to identify the client certificate.

      ```
      export NAME=client
      ```

   b. Run the following command to generate the client certificate.

      ```
      echo '{"CN":"'$NAME'","hosts":[""],"key":{"algo":"rsa","size":2048}}' |
      cfssl gencert -config=ca-config.json -ca=ca.pem -ca-key=ca-key.pem -
      hostname="$ADDRESS" - | cfssljson -bare $NAME
      ```

   This command creates three files in the current working directory: `client-key.pem`, `client.csr`, and `client.pem` files.

9. Run the following command to protect the client certificate with a password.

   ```
   openssl rsa -passout pass:<your_password> -aes256 -in client-key.pem -out
   client-ekey.pem
   ```

   Replace, *`<your_password>`* with a password for your client private key file. Remember the password that you provide as you'll have to provide it in the next step.

The `client-ekey.pem` file is created in the current working directory. You will need to provide the contents of the `client-ekey.pem` file and password in the next step.

10. In any text editor, create a JSON file which contains the contents of the `client-ekey.pem`, `client.pem`, and the password that you have used to protect the client certificate.

    The `client.pem` file contains the client certificate and the `client-ekey.pem` file contains the key.

    a. Copy the contents of the `client.pem`, the client public key file, as the value of the `cert` field.

    b. Copy the contents of the `client-ekey.pem`, the client private key file, as the value of the `key` field.

    c. Enter the password for the client private key file that you have provided in the previous step as the value of the `keyPassword` field.

    d. Replace all the new lines with the newline character `\n`.

    e. Create a JSON file with the edited values. The following code shows a sample JSON file. The sample values have been truncated with ellipses (...) for readability.

    ```
    {
    "cert":"-----BEGIN CERTIFICATE-----\nMIIDOjCC...\nBQAwD..jHPs=\n-----END
    CERTIFICATE-----",
    "key":"-----BEGIN RSA PRIVATE KEY-----\nProc-Type: 4,ENCRYPTED\nDEK-Info:
    AES-256-CBC,1870...\n\nNb...\n-----END RSA PRIVATE KEY-----",
    "keyPassword":"<your_password>"
    }
    ```

11. Validate and then save the JSON file. Remember the name of the JSON file as you have to provide the name of the file and its location. Let's consider that you save the JSON file as `etcdecred.json`.

3.6.2 Create a Kubernetes Secret for etcd

You must provide etcd credentials and etcd endpoints in the `values.yaml` file. MicroTx uses this information to establish a connection to etcd after the service is installed.

Before you begin, generate RSA certificates for etcd and create a JSON file with the contents of the generated certificates. See Generate RSA Certificates for etcd.

If you plan to deploy etcd and MicroTx within the same Kubernetes cluster, then it is optional for you to configure etcd with TLS. When etcd is configured with TLS, you must provide the certificate details in the `values.yaml` file for the transaction coordinator.

To create Kubernetes secret and Kubernetes configuration map:

1. Create a Kubernetes secret with the content available in the JSON file that you have created. Ensure that you create the Kubernetes secret in the namespace where you want to deploy MicroTx.

   ```
   kubectl create secret generic etcd-cert-secret \
       --from-file=location of etcdecred.json -n otmm
   ```

   Where,

- *etcd-cert-secret* is the name of the Kubernetes secret that you want to create. Note down this name as you will need to provide this name in the `YAML` file to install MicroTx.
- *location of etcdecred.json* is the location of the JSON file that you have created in the previous step.
- *otmm* is the namespace where you want to deploy MicroTx.

2. Create a configuration map for the `ca.pem` file, which you had created previously while initializing the certificate authority. Ensure that you create the configuration map in the namespace where you want to deploy MicroTx.

   ```
   kubectl create configmap etcd-ca-cert-map --from-file=location of ca.pem -
   n otmm
   ```

   Where,

   - *etcd-ca-cert-map* is the name of the configuration map that you want to create. Note down this name as you will have provide this name in the `values.yaml` file for MicroTx.
   - *location of ca.pem* is the location of the `ca.pem` file.
   - *otmm* is the namespace where you want to deploy MicroTx.

You will need to provide the etcd endpoints, certificate, Kubernetes secret, and Kubernetes configuration map that you have created in the `values.yaml` file. The following code snippet provides sample value which are based on the values used in the commands in this topic.

```
storage:
    type: etcd
    etcd:
      endpoints: "https://198.51.100.1:4002"
      skipHostNameVerification: "false"
      credentialSecret:
        secretName: "etcd-cert-secret"
        secretFileName: "etcdecred.json"
      cacertConfigMap:
        configMapName: "etcd-ca-cert-map"
        configMapFileName: "ca.pem"
```

If you do not provide the correct IP address for the `endpoints` field, then host verification fails when you install MicroTx. To bypass the host verification in development environments, you can set `skipHostNameVerification` to `true` in the `values.yaml` file of MicroTx.

Caution:

You must set the `skipHostNameVerification` field to `false` in production environments.

3.7 Set Up Oracle Database as Data Store

Create a data store for MicroTx to store the transaction logs. You can use either etcd or Oracle Database as the data store.

Skip this section if you want to set up etcd as data store. See Set Up etcd as Data Store.

Before installing MicroTx, you must install and configure a data store. Ensure that you set up the required networking rules to allow communication between the transaction coordinator and the data store.

For details about setting up the Oracle Database, refer to the documentation that is specific to the database that you want to set up.

Ensure that you have the required permissions to create tables in the database. When you install MicroTx, the service creates the required tables in the database. So MicroTx requires certain details about the database.

Topics:

- Grant Privilege to Run Stored Procedures
  The MicroTx coordinator creates a stored procedure when the service starts. If the Oracle Database user does not have the permission to create and run stored procedures, the service does not start.
- Get Autonomous Database Client Credentials
  MicroTx supports using Oracle Database as a persistent store to keep track of the transaction information.
- Create a Kubernetes Secret for Oracle Database Credentials
  MicroTx supports using Oracle Database as a persistent store to keep track of the transaction information.

3.7.1 Grant Privilege to Run Stored Procedures

The MicroTx coordinator creates a stored procedure when the service starts. If the Oracle Database user does not have the permission to create and run stored procedures, the service does not start.

Perform this task only if you use Oracle Database as the data store.

- To grant the privilege to create and run stored procedures, run the following command.

  Sample Command

  ```
  grant create procedure to <data_store_user>;
  ```

  Where, `<data_store_user>` is the name of the user who has access to the data store.

3.7.2 Get Autonomous Database Client Credentials

MicroTx supports using Oracle Database as a persistent store to keep track of the transaction information.

Skip this task if you are not using an Autonomous Database instance. If you are using an Autonomous Database instance, perform the following steps to get the Oracle client credentials (wallet files):

1. Download the wallet from the Autonomous Database instance. See Download Client Credentials (Wallets) in *Using Oracle Autonomous Database on Shared Exadata Infrastructure*.
   A ZIP file is downloaded to your local machine. Let's consider that the name of the wallet file is `Wallet_database.zip`.

2. Unzip the wallet file.

   ```
   unzip Wallet_database.zip
   ```

   The files are extracted to a folder. Note down the name of this folder. You will need to provide it in the next steps.

3. Create a configuration map to store the location of the folder where you have extracted the wallet files.
   Perform this step only if you want to deploy MicroTx in a Kubernetes cluster.

   Ensure that you create the configuration map in the namespace where you want to deploy MicroTx.

   ```
   kubectl create configmap db-wallet-configmap --from-file=/
   Wallet_database_folder/ -n otmm
   ```

   Where,

   - *db-wallet-configmap* is the name of the configuration map that you want to create. Note down this name as you will need to provide this name in the `values.yaml` file while deploying MicroTx.
   - *Wallet_database_folder* is the folder where you have extracted the contents of the zipped wallet file.
   - *otmm* is the namespace where you want to deploy MicroTx.

   Replace these values with values that are specific to your environment.

4. Perform the following steps only if you want to deploy MicroTx in Docker Swarm.

   a. Create 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:

      ```
      <publicIP>:<portNumber>/<database unique name>.<host domain name>
      ```

      For example, `123.213.85.123:1521/CustDB_iad1vm.sub05031027070.customervcnwith.oraclevcn.com`.

   b. Append `&wallet_location=/app/Wallet` to the connection string that you have created in the previous step. For example:

      ```
      tcps://adb.us-ashburn-1.oraclecloud.com:1522/
      bfeldfxbtjvtddi_brijeshadw1_medium.adb.oraclecloud.com?
      retry_count=20&retry_delay=3&wallet_location=/app/Wallet
      ```

      Where, `/app/Wallet` is the location where you have downloaded the wallet file.

      Note down this connection string as you'll have to provide this value later in the `tcs-docker-swarm.yaml` file.

Next, based on the environment in which you want to install MicroTx, create a Docker secret or Kubernetes secret to provide the Oracle Database login details.

3.7.3 Create a Kubernetes Secret for Oracle Database Credentials

MicroTx supports using Oracle Database as a persistent store to keep track of the transaction information.

You must provide the Oracle Database credentials in the `values.yaml` file. MicroTx uses the credentials to establish a connection to the database after the service is installed.

If you are using an Autonomous Database instance, ensure that you have downloaded the wallet and created a configuration map before you begin with the following steps. See Get Autonomous Database Client Credentials.

To create a Kubernetes secret to provide the Oracle Database login details:

1. Create a Kubernetes secret with the Oracle Database login details. Ensure that you create the Kubernetes secret in the namespace where you want to deploy MicroTx.

   The following command creates a Kubernetes secret with the name `db-secret` in the `otmm` namespace with the password of user `acme`. When you run this command in your environment, replace these values with values specific to your environment.

   ```
   kubectl create secret generic db-secret \
       --from-literal=secret='{"password":"*****", "username":"acme"}' -n otmm
   ```

2. Note down the name of the Kubernetes secret that you have created. You will need to provide this name in the `values.yaml` file while deploying MicroTx.

Update the `values.yaml` with the name of the Kubernetes secret that you have created to store the Oracle Database credentials and the connection string. Additionally, provide the name of the configuration map if you are using an Autonomous Database instance.

4

Install on a Kubernetes Cluster

You can install Transaction Manager for Microservices (MicroTx) on Docker or on a Kubernetes cluster.

If you want to install MicroTx on a Kubernetes cluster, skip this section and see Install on Docker Swarm.

Before you begin, ensure that you have completed the prerequisites. See Prepare.

Perform the following steps to install MicroTx:

1. Push Images to a Remote Docker Repository
2. Configure the values.yaml File
3. Install MicroTx

- Push Images to a Remote Docker Repository
  The installation bundle that you have downloaded to your local system contains a Docker image of MicroTx and the MicroTx console.
- Configure the values.yaml File
  The installation bundle contains `values.yaml` file, the manifest file of the application, which contains the deployment configuration details for MicroTx.
- Install MicroTx
  Use Helm to install MicroTx on a Kubernetes cluster.
- Access MicroTx
  To access MicroTx, 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.
- Check the Coordinator Health
  After installing MicroTx, run the following command to check the health of the coordinator and to validate that the installation was completed successfully.

4.1 Push Images to a Remote Docker Repository

The installation bundle that you have downloaded to your local system contains a Docker image of MicroTx and the MicroTx console.

Load these image to your local repository, and then push the images to a remote Docker repository. Kubernetes pulls these images from the remote repository to install MicroTx and the MicroTx console. Use the console to manage the transactions.

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.

Perform the following steps to push the Docker images of MicroTx and console to a remote Docker repository:

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. Load the images to the local Docker repository.

   a. Load the MicroTx image to the local Docker repository. The MicroTx image is located at `installation_directory/otmm-RELEASE/otmm/image/tmm-RELEASE.tgz`.

   ```
   cd installation_directory/otmm-RELEASE/otmm
   docker load < image/tmm-RELEASE.tgz
   ```

   The following message is displayed when the image is loaded.

   ```
   Loaded image: tmm:RELEASE
   ```

   b. Load the console image to the local Docker repository. The console image is located at `installation_directory/otmm-RELEASE/console/image/Console-RELEASE.tgz`.

   ```
   cd installation_directory/otmm-RELEASE/console
   docker load < image/Console-RELEASE.tgz
   ```

   The following message is displayed when the image is loaded.

   ```
   Loaded image: Console:RELEASE
   ```

3. Use the following commands to specify a unique tag for the images 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.

   Sample Commands

   ```
   docker tag tmm:RELEASE <region-key>.ocir.io/otmmrepo/tmm:RELEASE
   docker tag console:RELEASE <region-key>.ocir.io/otmmrepo/console:RELEASE
   ```

Where, `<region-key>.ocir.io/otmmrepo` is the remote Docker registry to which you want to push the image files, `tmm:`*`RELEASE`* and `console:`*`RELEASE`*. Provide the registry details based on your environment.

4. Push the Docker images from your local repository to the remote Docker repository.

   Syntax

   ```
   docker push remote_image_tag
   ```

   Sample Commands

   ```
   docker push <region-key>.ocir.io/otmmrepo/tmm:RELEASE
   docker push <region-key>.ocir.io/otmmrepo/console:RELEASE
   ```

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.

4.2 Configure the values.yaml File

The installation bundle contains `values.yaml` file, the manifest file of the application, which contains the deployment configuration details for MicroTx.

Replace the sample values in the `values.yaml` file to provide the environment details, image details, and configuration details to deploy MicroTx.

While deploying MicroTx to a Kubernetes cluster, Helm pulls the MicroTx image from the remote Docker registry. In the `values.yaml` file, specify the image to pull and the credentials to use when pulling the images.

To provide configuration details for MicroTx:

1. Open one of the following `values.yaml` file in any code editor. These files contain sample values. Select a file based on the container platform that you are using, whether you want to use an Istio service mesh, and whether you want to deploy the MicroTx Free or Enterprise edition.

   The following table provides the relative path of the `values.yaml` file from *`installation_directory`*`\otmm-`*`RELEASE`*`\otmm\helmcharts` folder.

File Path	Container Platform	Description
`.\tmm\ee`	Kubernetes	Deploys MicroTx Enterprise edition in Istio service mesh.
`.\tmm`	Kubernetes	Deploys MicroTx Free in Istio service mesh.
`.\tmmk8s\ee`	Kubernetes	Deploys MicroTx Enterprise edition without using Istio service mesh.
`.\tmmk8s`	Kubernetes	Deploys MicroTx Free without using Istio service mesh.
`.\tmm-istio-openshift\ee`	OpenShift	Deploys MicroTx Enterprise edition in Istio service mesh.
`.\tmm-istio-openshift`	OpenShift	Deploys MicroTx Free in Istio service mesh.
`.\tmm-openshift\ee`	OpenShift	Deploys MicroTx Enterprise edition without using Istio service mesh.
`.\tmm-openshift`	OpenShift	Deploys MicroTx Free without using Istio service mesh.

2. Replace the sample values with values that are specific to your environment.

   The tables in this section describe the properties for the environment, storage, authorization, authentication, and other configuration details that are required to install MicroTx.

3. Save your changes.

Topics

- Environment Details
  In the `values.yaml` file, provide information about the environment details in which you want to install MicroTx.
- Image Properties
  Under `tmmImage`, provide information about the MicroTx Docker image. It is mandatory to provide values for these properties.
- Transaction Coordinator Properties
  Under `tmmConfiguration`, provide information to configure MicroTx.
- Data Store Properties
  MicroTx uses a data store for persistence of the transaction state and to store the transaction logs.
- Istio Details
  If you are using an Istio service mesh, provide details about the Istio ingress gateway that you have set up in the `values.yaml` file.
- Caching Properties
  You can enable caching to improve performance. It optimizes the read and write operations for the transaction logs that are stored in etcd or Oracle Database.
- Logging Properties
  Under `logging`, provide details for the MicroTx logs.
- Metrics Property
  Under `metrics`, enable Prometheus to scrape the metrics logs of the MicroTx coordinator.
- Authorization Properties
  MicroTx supports authorization across participant services and coordinator by propagating the JWT token in every request.
- Authentication Properties
  Under `authentication`, enable JSON Web Token (JWT) authentication.
- Identity Provider Properties
  Under `identityProvider`, enter property values for the JSON Web Token (JWT) which the MicroTx coordinator uses for authentication.
- Encryption Key Properties
  Under `encryption`, specify the encryption key that MicroTx uses to encrypt the access and refresh tokens. You must provide values for these properties if you have enabled `authTokenPropagationEnabled` under `tmmConfiguration.authorization`.
- Transaction Token Properties
  Under `transactionToken`, specify the key pair that you want to use for transaction token.
- Console Configuration Properties
  Under `tmmConsoleConfiguration`, specify the properties for the MicroTx console.

4.2.1 Environment Details

In the `values.yaml` file, provide information about the environment details in which you want to install MicroTx.

Property	Description
`applicationNameSpace`	Specify the namespace in which you want to deploy MicroTx. For example, `otmm`.
`tmmReplicaCount`	Specify the number of MicroTx pod replicas that you want to create. Oracle recommends a minimum of 3 replicas for production environments. Development or test environments may be able to use fewer replicas.

4.2.2 Image Properties

Under `tmmImage`, provide information about the MicroTx Docker image. It is mandatory to provide values for these properties.

Property	Description
`image`	Enter the tag of the MicroTx image that you have pushed to the remote repository. For example, `oracle-tmm:`*`RELEASE`*. See Push Images to a Remote Docker Repository.
`imagePullPolicy`	Enter `Always` to ensure that the image is pulled during the installation.
`imagePullSecret`	Specify the name of the Kubernetes secret that you have created. This secret is used to pull the Docker images from the remote repository. For example, `regcred`. See Create a Kubernetes Secret to Access Docker Registry.

The following code snippet provides sample values for

`tmmImage`

.

```
tmmImage:
  image: oracle-tmm:RELEASE
  imagePullPolicy: Always
  imagePullSecret: regcred
```

4.2.3 Transaction Coordinator Properties

Under `tmmConfiguration`, provide information to configure MicroTx.

Property	Description
`tmmAppName`	Enter the name of the MicroTx application that you want to create. When you install MicroTx, Helm creates the MicroTx application with the name that you specify. Note down this name as you will need to provide it later. For example, `tmm-app`.

Property	Description
`tmmid`	Enter a value to uniquely identify each instance of MicroTx that you install. The unique identifier must have 5-characeters and can contain only alphanumeric characters (a-z, A-Z, and 0-9). For example, `TMM01`. Use this ID to identify MicroTx when there are multiple installations. You cannot use this ID to differentiate between replicas of a single instance of MicroTx installation as all the replicas have the same ID. You can't change this value after installing MicroTx.
`port`	Enter the port over which you want to internally access MicroTx within the Kubernetes cluster where you will install this service. Create the required networking rules to permit inbound and outbound traffic on this port. Note down this number as you will need to provide it later. For example, `9000`.
`tmmExternalURL`	Enter the details required to create the external URL to access MicroTx from outside the Kubernetes cluster where you have deployed the service. Enter the protocol, host, and port number to access the Istio ingress gateway. See Access MicroTx.
`httpClientTimeoutInSecs`	Specify the maximum amount of time, in seconds, for which the HTTP callback API requests sent by the MicroTx coordinator to the participant services remains active. Enter an integer between 0 to 900. The default value is 180 seconds and the maximum value is 900 seconds. If you set this value to 0, then MicroTx does not enforce any limit. When the coordinator sends a HTTP callback API request to the participant services, the participant services must respond within the time period that you specify. If the participant service does not respond within the specified time period, the HTTP request sent by the coordinator times out.
`xaCoordinator`, `lraCoordinator`, or `tccCoordinator`	Set `enabled: "true"` for the transaction protocols that your microservices use. MicroTx supports three distribution transaction protocols: XA, Saga, and TCC. If you want to nest an XA transaction within an Saga transaction, set `enabled: "true"` for both `xaCoordinator` and `lraCoordinator`.
`txMaxTimeout`	Only for the XA transaction protocol. 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 is 600000 ms.
`narayanaLraCompatibilityMode`	Only for the Saga transaction protocol. Set `enabled` to `true` only when you want to use Saga participant applications that were implemented to work with the Narayana LRA Coordinator and now would participate in Saga transactions using MicroTx. Enable this mode to ensure that the MicroTx Saga APIs return the same response data that Narayana LRA Coordinator APIs return.
`maxRetryCount`	The maximum number of times that the transaction coordinator retries sending the same request again in case of any failures. For example, 10.
`minRetryInterval`	The minimum interval, in milliseconds, after which the transaction coordinator retries sending the same request again in case of any failures. The default value is 1000 ms.
`maxRetryInterval`	The maximum retry interval, in milliseconds, before which the transaction coordinator retries sending the same request again in case of any failures. For example, 10000.

Property	Description
`skipVerifyInsecureTLS`	Oracle recommends that you set this value to `false` and set up a valid certificate signed by trusted authorities for secure access. When you set this value to `false`, the transaction coordinator accesses the participant applications over the HTTPS protocol with a valid certificate signed by trusted authorities. The default value is `false`. If you set this value to `true`, the transaction coordinator can access the participant application's callback URL, without a valid SSL certificate, in an insecure manner. **Caution:** Do not set this value to `true` in production environments.
`countersUpdateInterval`	Enter a value between 120 to 1800 seconds to specify data collection time interval or the rate at which information about the most recent transactions is updated on the MicroTx console. If you enter 300 seconds, information about the most recent transactions is updated every 300 seconds on the console. The default value is 120 seconds. You can also update this value in the MicroTx console by changing the value in the **Interval** box.

4.2.4 Data Store Properties

MicroTx uses a data store for persistence of the transaction state and to store the transaction logs.

You can use an etcd cluster, Oracle Database, or internal memory to store the transaction logs. When you want to use multiple replicas of the transaction coordinator, you must set up an etcd cluster or Oracle database as the transaction log data store. If you use internal memory, you can't create multiple replicas of the transaction coordinator. Use internal memory only for development environments as all the transaction details are lost every time you restart MicroTx.

Type of Data Store for Transaction Logs

Under `tmmConfiguration.storage`, specify the type of data store that MicroTx uses for persistence of transaction state. After specifying the type of data store, you can provide additional details to connect to the external data store.

Property	Description
`type`	Enter one of the following values to specify the persistent data that you want MicroTx to use to track the transaction information. • `etcd` to use etcd as the data store. You must provide details to connect to the etcd data store in the `storage.etcd` field. • `db` to use Oracle Database as the data store. You must provide details to connect to the Oracle data store in the `storage.db` field. • `memory` to skip entering details to connect to either etcd or Oracle Database and use the internal memory instead. When you use internal memory, all the transaction details are lost every time you restart MicroTx.

Property	Description
`completedTransactionTTL`	The time to live (TTL) in seconds for a completed transaction record in the transaction data store. The permissible range of values is 60 to 1200 seconds. When the specified time period expires, the completed transaction entry is removed from the data store.

Oracle Database as Data Store for Transaction Logs

Under `tmmConfiguration.storage.db`, specify the details to connect to an Oracle Database. Skip this section and do not provide these values if you are connecting to an etcd database or using internal memory.

The following table specifies the properties required to connect to an Oracle Database as the MicroTx data store.

Property	Description

`connectionString`	Enter the connection string for the data store in Oracle 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:
	`tcps://<database-host>:<database-portNumber>/`
	`<service_name>`
	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 credential wallet. See Download Client Credentials (Wallets) in *Using Oracle Autonomous Database on Shared Exadata Infrastructure*.
	For example:
	`tcps://adb.us-phoenix-1.oraclecloud.com:1521/`
	`sales.adb.oraclecloud.com`
	Additionally, you can pass database timeout parameters, such as `retry_count` and `retry_delay`, by appending it to the connection string. For example,
	`<connection-string>?retry_count=20&retry_delay=3`
	• 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:
	`<database-host-IPaddress>:<database-portNumber>/`
	`<databaseUniqueName>.<hostDomainName>`
	For example:
	`123.213.85.123:1521/`
	`CustDB_iad1vm.sub05031027070.customervcnwith.oracl`
	`evcn.com`
	Where, `CustDB_iad1vm.sub05031027070.customervcnwith` is the unique name of the database or the service name.
	• If you are using Oracle RAC database, see About Connecting to an Oracle RAC Database Using SCANs.

`credentialSecretName`	Enter the name of the Kubernetes secret that contains the credentials to connect to the Oracle Database. Example, `db-secret`. See Create a Kubernetes Secret for Oracle Database Credentials.
`connectionParams`	Enter a space-separated list of connection parameters. For details about the connection parameters that you can specify, see https://pkg.go.dev/github.com/godror/godror#section-documentation. For example, `poolMinSessions=1 poolMaxSessions=300`

Property	Description
`walletConfigMap.configMapName`	Provide a value for this field only if you connect to a database that uses a credential wallet, such as Autonomous Database. For other databases, you can leave this field empty. Enter the name of the configuration map that you have created for the credential wallet of the Autonomous Database instance. See Get Autonomous Database Client Credentials. For example, `db-wallet-configmap`.

etcd Database as Data Store for Transaction Logs

Under `tmmConfiguration.storage.etcd`, specify the details to connect to an etcd database. Skip this section and do not provide these values if you are connecting to an Oracle database or are using internal memory.

The following table specifies the properties required to connect to an etcd database as the MicroTx data store.

Before you provide this information, ensure that you complete the following tasks and note down the required details.

1. Generate RSA Certificates for etcd
2. Create a Kubernetes Secret for etcd

Property	Description
`endpoints`	Enter the external IP address of the etcd database server. If you have installed the etcd cluster in the Kubernetes cluster where you will install MicroTx, then provide the Kubernetes service name and the port of the etcd cluster (nodes) as values. Otherwise, enter a comma-separated list of host names or IP addresses of the etcd cluster nodes along with the ports, such as `198.51.100.1:4002,198.51.100.2:4002,198.51.100.3:4002`.
`skipHostNameVerification`	Set this to `false` to verify the IP address of the etcd database server. If you set this to `true`, then the server host name or IP address is not verified. You can set this field to `true` only for test or development environments. **Caution:** You must set this field to `false` in production environments.
`credentialSecret.secretName`	Enter the path to the Kubernetes secret in the container. The secret contains the client credentials, client key, and the password that you have used to protect the client certificate. For example, `/etc/otmm/etcd-cert-secret`. See Create a Kubernetes Secret for etcd.
`credentialSecret.secretFileName`	Enter the location of the JSON file, that contains client credentials, client key, and the password that you have used to protect the client certificate. For example, `/etc/otmm/etcdecred.json`. See Generate RSA Certificates for etcd.
`cacertConfigMap.configMapName`	Enter the name of the configuration map file, which you had created while initializing the certificate authority. For example, `etcd-ca-cert-map`. See Create a Kubernetes Secret for etcd.

Property	Description
`cacertConfigMap.configMapFileName`	Enter the name of the PEM file that you had created while initializing the certificate authority. For example, `ca.pem`. See Generate RSA Certificates for etcd.

The following code snippet provides sample value which are based on the sample values provided in the Generate RSA Certificates for etcd and Create a Kubernetes Secret for etcd topics.

```
storage:
    type: etcd
    etcd:
      endpoints: "https://198.51.100.1:4002"
      skipHostNameVerification: "false"
      credentialSecret:
        secretName: "etcd-cert-secret"
        secretFileName: "etcdecred.json"
      cacertConfigMap:
        configMapName: "etcd-ca-cert-map"
        configMapFileName: "ca.pem"
```

If you do not provide the correct IP address for the `endpoints` field, then host verification fails when you install MicroTx.

4.2.5 Istio Details

If you are using an Istio service mesh, provide details about the Istio ingress gateway that you have set up in the `values.yaml` file.

Skip providing values for the properties mentioned in the following table if you aren't using an Istio service mesh. Instead provide a value for the property. For example, `otmm-tcs-otmm.apps-crc.testenv`.
Before you begin, ensure that you have set up and configured Istio. See Install the Required Software for Kubernetes.

Property	Description
`ingressHostName`	Enter the name of the OpenShift ingress host. For example, `otmm-tcs-otmm.apps-crc.testing`. Use this property only if you want to deploy MicroTx in OpenShift platform. Refer to the `values.yaml` file, the Helm Charts available in the `*installation_directory*/otmm-*RELEASE*/otmm/helmcharts/tmm-istio-openshift` folder. This sets up a FQDN that is resolvable and it configures an ingress resource and route to the specified OpenShift ingress host. See https://docs.openshift.com/container-platform/4.14/rest_api/network_apis/ingress-networking-k8s-io-v1.html#spec-rules-2.

Property	Description
`istioSystemNameSpace`	The namespace in which you have installed Istio. The default namespace is `istio-system`. If you have installed Istio in another namespace, run the following command to find all the namespaces in the cluster. `kubectl get ns`
`istioIngressGateway.name`	Enter the name of the Istio ingress gateway that you have created. For example, `ingressgateway`. To find the name of the Istio ingress gateway, run the following command and from the response note down the value for the `istio` label. `kubectl describe service/istio-ingressgateway -n istio-system`
`istioIngressGateway.tlsEnabled`	Set this to `true` to enable access to the service using the HTTPS protocol. When you set this to `true`, you must provide details of the Kubernetes secret that contains the SSL key and certificate to access Istio using HTTPS. **Note:** You must set this value to `true` in production environments.
`istioIngressGateway.credentialName`	You must specify a value for this property if `tlsEnabled` is set to `true`. Enter the name of the Kubernetes secret that you have created to enable access to Istio using the HTTPS protocol. See Create a Kubernetes Secret with SSL Details for Istio. For example, `tls-credential`.
`istioIngressGateway.hosts`	Enter the external IP Address of Istio ingress gateway or the name of the hosts. If you are using a load balancer or have multiple hosts, enter a comma-sepaeated list of host names or IP addresses. See Find IP Address of Istio Ingress Gateway.

Sample Property Values for Istio

The following code snippet provides sample property values for Istio.

```
istioSystemNameSpace: istio-system
istioIngressGateway:
  name: ingressgateway
  tlsEnabled: "true"
  credentialName: tls-credential
  hosts: 192.0.2.1
```

4.2.6 Caching Properties

You can enable caching to improve performance. It optimizes the read and write operations for the transaction logs that are stored in etcd or Oracle Database.

Property	Description
`enabled`	Enter `true` to enable local caching for the coordinator. You can set this to `true` only if you use etcd cluster or Oracle Database to store the transaction information. You can not enable caching if you use internal memory as the transaction store.
`maxCapacity`	Enter a decimal value as the maximum storage capacity of the cache in GB. For example, `1.5`. The default value is `1`. Ensure that your environment has the required memory to support the maximum value that you enter.

When you enable caching, you must also enable session affinity for the coordinator. See Enable Session Affinity.

The following code snippet provides sample values for the `caching` property in the `values.yaml` file.

```
caching:
  enabled: true
  maxCapacity: 1.5
```

4.2.7 Logging Properties

Under `logging`, provide details for the MicroTx logs.

Property	Description
`level`	Enter one of the following types to specify the log level for MicroTx: • `info`: Logs events that occur during the normal operation of the MicroTx. 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 to indicate that there is an issue that requires troubleshooting. • `debug`: Logs all the events. Use this setting when you want to debug an issue.
`httpTraceEnabled`	Set this to `true` to log all the HTTP request and responses in MicroTx when you want to debug. When you set this value as `true`, you must also set the `level` as `debug`.

The following code snippet provides sample values for the `logging` property in the `values.yaml` file.

```
logging:
  level: debug
  httpTraceEnabled: true
```

4.2.8 Metrics Property

Under `metrics`, enable Prometheus to scrape the metrics logs of the MicroTx coordinator.

Property	Description
`enabled`	Set this to `true` so that Prometheus can scrape the metrics logs of the MicroTx coordinator. By default, this is set this to `true`.

The following code snippet provides sample values for the `metrics` property in the `values.yaml` file.

```
metrics:
  enabled: true
```

4.2.9 Authorization Properties

MicroTx supports authorization across participant services and coordinator by propagating the JWT token in every request.

Under `tmmConfiguration.authorization`, use the `authTokenPropagationEnabled` field to control this function. Configure your identity providers to auto-refresh the expired access tokens at the coordinator.

Property	Description
`enabled`	Set this to `true` to enforce RBAC rules on the MicroTx coordinator API access. The first step in authorization is to enforce RBAC controls at the MicroTx coordinator. **Caution:** You must set this field to `true` in production environments.
`authTokenPropagationEnabled`	Set this to `true` to enable token propagation to ensure secure communication between the participant services and MicroTx. MicroTx performs the following actions: • The MicroTx libraries propagate the authorization headers in every outgoing call to the MicroTx coordinator. The coordinator uses the propagated access token from incoming requests for the authorization checks. • The MicroTx coordinator encrypts and stores the access token and refresh token details in the transaction store. These tokens are appended in the callback API calls from the MicroTx coordinator to the MicroTx library that is present in the participant application. When you enable token propagation, you must provide the details for the encryption keys under the `encryption` property in the `values.yaml` file.

4.2.10 Authentication Properties

Under `authentication`, enable JSON Web Token (JWT) authentication.

Property	Description
`requestsWithNoJWT`	Enter `DENY` to enable authorization checks at coordinator, such as role based access control (RBAC). This ensures that each request has a JWT token. Enter `ALLOW` to bypass JWT authentication. This permits requests that do not have JWT tokens. **Caution:** You must set this field to `DENY` in production environments. When set to `DENY`, you must provides value for the identity provider properties. See Identity Provider Properties.

The following code snippet provides sample values for the `authentication` field in the `values.yaml` file.

```
authentication:
  requestsWithNoJWT: DENY
```

4.2.11 Identity Provider Properties

Under `identityProvider`, enter property values for the JSON Web Token (JWT) which the MicroTx coordinator uses for authentication.

When you set `authentication.requestsWithNoJWT` to `DENY`, you must provides values for all the identity provider properties listed in the table below.

However, provide values for the `audience`, `adminUserRoles`, `adminUserRolesPath`, and `clientSecretName` properties to ensure that users can access the MicroTx console. If you don't want to provide access to the MicroTx console, you can skip providing values for these properties.

Property	Description
`serverType`	Enter `idcs` if you are using Oracle IDCS as the identity provider. Otherwise, enter `other`. The default value is `other`.
`scopes`	If the `serverType` is `idcs`, you *must* specify a scope to grant different levels of access. If you are not using Oracle IDCS as the identity provider, do not provide a value for this property. For Oracle IDCS, enter a space-separated list of scopes. The default scope for Oracle IDCS is `openid groups`.
`issuer`	Identifies the JWT token issuer. Enter the URI of the identity server that you have set up. It is the value of the `issuer` field in the Discovery URL. For example, `https://identity.oraclecloud.com`. See Run the Discovery URL.
`jwksUri`	The URL of the identity provider's publicly hosted `jwksUri`, which is used to validate signature of the JWT. The JSON Web Key Set (JWKS) contains the cryptographic keys which are used to verify the incoming JWT tokens. See Run the Discovery URL.

Property	Description
`identityProviderUrl`	Specify the URL of the JWT identity provider. This information is required to create a new access token by using the refresh token. If you do not provide this information, expired access tokens are not auto-refreshed. For example, `http://192.0.2.1:8080/auth/realms/tmmdev` when you use Keycloak as the identity provider. See Run the Discovery URL.
`audience`	Enter the audience of the token. Every JWT is validated to check the audience. You must provide a value for this parameter to access the MicroTx console. Note down this value from the JWT access token.
`adminUserRoles`	Enter a comma-separated list of names of the administrator roles that you have configured in the identity provider to grant access to the MicroTx administrator APIs. Only the users that are granted this role are permitted to access the console. For example, `admin, consoleadmin`. See Specify the Admin Role in YAML file.
`adminUserRolesPath`	Enter the path to the administrator role in the JWT token. For example, `realm_access, roles`. See Specify the Admin Role in YAML file.
`clientSecretName`	Enter the name of the Kubernetes secret that you have created to ensure that users can access the MicroTx console. See Create a Secret with Identity Provider Client Credentials.

The following code snippet provides sample values for the `authentication` field in the `values.yaml` file. The sample values in this example are based on Run the Discovery URL and Specify the Admin Role in YAML file.

```
identityProvider:
    issuer: "https://identity.oraclecloud.com"
    jwksUri: "https://idcs-a83e4...identity.oraclecloud.com:443/admin/v1/
SigningCert/jwk"
    identityProviderUrl: "https://idcs-a83e4...identity.oraclecloud.com/
oauth2/v1/token"
    audience: "account"
    adminUserRoles: "admin"
    adminUserRolesPath: "userAppRoles"
    clientSecretName: "console-identity-client-secret"
```

The example tenant base URL, `https://idcs-a83e4...identity.oraclecloud.com`, has been truncated with ellipses (...) for readability. Copy the complete value in your environment.

4.2.12 Encryption Key Properties

Under `encryption`, specify the encryption key that MicroTx uses to encrypt the access and refresh tokens. You must provide values for these properties if you have enabled `authTokenPropagationEnabled` under `tmmConfiguration.authorization`.

Property	Description
`encryptionSecretKeyVersion`	Specify the version of the secret key that you want to use for encrypting the transaction tokens from the list of secret keys that you provide.
`encryptionSecretKeys.secretKeys.SecretKeyName`	Specify the name and version of the Kubernetes secrets that contain encryption key as the value. To support the encryption keys rotation, you can specify multiple encryption keys and their versions.

Property	Description
`encryptionSecretKeys.secretKeys.version`	Enter a natural number as the version of the Kubernetes secret. Ensure that the version is unique for each secret key.

If you create a new Kubernetes secret key, do not delete the entry for the previous secret key immediately. You may delete the old key and the corresponding entry in the `values.yaml` file after a few days because existing transactions may be using the older versions of the key. After a few days, you can update the `values.yaml` file, and then update MicroTx.

The following code snippet provides sample values for the `encryption` field in the `values.yaml` file. The sample values in this example are based on the values used in the sample commands in Generate a Kubernetes Secret for an Encryption Key.

```
encryption:
  encryptionSecretKeyVersion: "1"
  encryptionSecretKeys:
    secretKeys:
      - secretKeyName: "encryption-secret-key1"
        version: "1"
      - secretKeyName: "encryption-secret-key2"
        version: "2"
```

4.2.13 Transaction Token Properties

Under `transactionToken`, specify the key pair that you want to use for transaction token.

If you set `transactionTokenEnabled` to `true`, it is mandatory to provide values listed in the following table.

Property	Description
`transactionTokenEnabled`	Set this to `true` when you want MicroTx to include a signed transaction token, `tmm-tx-token`, in the request header. You don't have to create the `tmm-tx-token` transaction token or pass it in the request header. The MicroTx library creates this token based on the private-public key pair that you provide. For more information about creating the key pair, see Create a Key Pair for Transaction Token.
`transactionTokenKeyPairVersion`	Enter the version of the key pair that you want to use for signing and verification of the transaction token. When there are multiple key pairs, you must specify the version of the key pair that you want to use.
`transactionTokenKeyPairs.keyPairs.privateKeyName`	Enter the name of the Kubernetes secret which has the base64-encoded value of the private key.
`transactionTokenKeyPairs.keyPairs.publicKeyName`	Enter the name of the Kubernetes secret which has the base64-encoded value of the public key.
`transactionTokenKeyPairs.keyPairs.privateKeyPasswordName`	Enter the name of the Kubernetes secret which has the value of the pass phrase that you had provided while generating the private key.
`transactionTokenKeyPairs.keyPairs.version`	Enter the version of the private-public key pair that you want to use.

The following code snippet provides sample values for the `transactionToken` field.

```
transactionToken:
  transactionTokenEnabled: "true"
  transactionTokenKeyPairVersion: "1"
  transactionTokenKeyPairs:
    keyPairs:
      - privateKeyName: "TMMPRIVKEY1"
        publicKeyName: "TMMPUBKEY1"
        privateKeyPasswordName: "TMMPRIVKEYPASSWD1"
        version: "1"
      - privateKeyName: "TMMPRIVKEY2"
        publicKeyName: "TMMPUBKEY2"
        privateKeyPasswordName: "TMMPRIVKEYPASSWD2"
        version: "2"
```

4.2.14 Console Configuration Properties

Under `tmmConsoleConfiguration`, specify the properties for the MicroTx console.

Ensure that you have completed the following tasks before you provide the property values.

- Push the MicroTx console image to the remote Docker repository. See Push Images to a Remote Docker Repository.
- Set up an identity provider. See Set Up Oracle Identity Providers.
- Set permissions for administrators. Administrators can access the MicroTx 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. See Specify the Admin Role in YAML file and Create a Secret with Identity Provider Client Credentials.
- Create a Kubernetes secret to store the cookie encryption password. See Create a Secret with Cookie Encryption Password for Kubernetes

It is mandatory to provide values for all the properties specified in the following table.

Property	Description
`tmmConsoleAppName`	Name of the MicroTx application that you want to create. Helm creates the console application with the name that you specify. Provide a unique name to identify the application. The name must start and end with an alphanumeric character, be between 4 to 63 characters, and contain only lowercase characters. Optionally, you can use hyphen (-) and dot (.) as special characters. For example, `otmm-console`.
`tmmConsoleImage.image`	Enter the path of the console image in the remote repository. For example, `iad.ocir.io/mytenancy/Console:`*`RELEASE`*. If you don't provide a value, the console application is not deployed.
`tmmConsoleImage.imagePullPolicy`	Enter `Always` to ensure that the image is pulled during the installation.
`tmmConsoleImage.imagePullSecret`	Specify the name of the Kubernetes secret that you want to use to pull the Docker images. See Create a Kubernetes Secret to Access Docker Registry. For example, `regcred`.
`port`	Specify a port number. Ensure that the required networking rules are set up to ensure that traffic is permitted to and from this port to the MicroTx console.

Property	Description
`cookieEncryptionPasswordSecretName`	Specify the name of the Kubernetes secret that you have created to store the cookie encryption password for the Console. See Create a Secret with Cookie Encryption Password for Kubernetes. For example, `console-cookie-encryption-password-secret`.

The following code snippet provides sample values for `tmmConsoleConfiguration`.

```
tmmConsoleConfiguration:
  tmmConsoleAppName: otmm-console
  # If tmmConsoleImage.image is empty, the console application is not
deployed.
  tmmConsoleImage:
    image: iad.ocir.io/mytenancy/Console:RELEASE
    imagePullPolicy: Always
    imagePullSecret: regcred
  port: 8080
  cookieEncryptionPasswordSecretName: "console-cookie-encryption-password-
secret"
```

4.3 Install MicroTx

Use Helm to install MicroTx on a Kubernetes cluster.

1. Navigate to the `helmcharts` folder for MicroTx.

   ```
   cd installation_directory/otmm-RELEASE/otmm/helmcharts
   ```

2. Deploy MicroTx using the configuration details provided in the `values.yaml` file. Run one of the following commands based on whether you want to install only the coordinator or you want to install the console as well.

   Syntax

   ```
   helm install <release name> --namespace <namespace> <chart directory> --
   values <values.yaml>
   ```

   Example

   - Use the following command to install MicroTx as an application named `tmm-app` in the `otmm` namespace.

     ```
     helm install tmm-app --namespace otmm tmm/ --values tmm/values.yaml
     ```

   - Use the following command to install the MicroTx console and MicroTx as an application named `tmm-app` in the `otmm` namespace.

     ```
     helm install tmm-app --namespace otmm tmm/ --values tmm/ee/values.yaml
     ```

   Where,

- `tmm-app` is the name of the application that you want to create.
- `otmm` is the namespace in Kubernetes cluster, where you want to install MicroTx.
- *`installation_directory`*`/otmm-`*`RELEASE`*`/otmm/helmcharts/tmm` is the folder that contains the `chart.yaml` file for MicroTx.
- *`installation_directory`*`/otmm-`*`RELEASE`*`/otmm/helmcharts/tmm/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 MicroTx. It does not contain properties that are specific to the MicroTx console.
- *`installation_directory`*`/otmm-`*`RELEASE`*`/otmm/helmcharts/tmm/ee/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 the MicroTx coordinator as well as the MicroTx console.

The following message is displayed.

```
NAME: otmm
LAST DEPLOYED: Tue Apr 19 21:14:25 2022
NAMESPACE: otmm
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

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 `otmm` and their status.

```
kubectl get all -n otmm
```

Sample response

Some of the values may be truncated with ... for the sake of readability. When you run this command in your environment, you will see the entire value.

```
NAME              READY   STATUS    RESTARTS   AGE
pod/otmm-tcs-0    2/2     Running   0          38s

NAME               TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
service/otmm-tcs   ClusterIP   10.110........   <none>        9000/TCP
38s

NAME                          READY   AGE
statefulset.apps/otmm-tcs     1/1     38s
```

After the installation is complete, you can access MicroTx.

The next chapter provides instructions to install and run sample applications in your environment. See Deploy Sample Applications.

4.4 Access MicroTx

To access MicroTx, 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. You will use different URLs depending on whether you want to access MicroTx from within the Kubernetes cluster where you have deployed the service or from a different Kubernetes cluster.

Internal URL to access MicroTx

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

To access MicroTx, 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. For example, `tmm-app`.
- *`internalPort`*: Port number that you have entered for the `port` property in the `values.yaml` of the MicroTx. 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 URL is `https://tmm-app:9000/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

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

To access MicroTx 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 URL is `https://192.0.2.1:443/api/v1`.

4.5 Check the Coordinator Health

After installing MicroTx, run the following command to check the health of the coordinator and to validate that the installation was completed successfully.

```
curl https://externalHostname:externalPort/health
```

To identify values for the *`externalHostname`* and *`externalPort`*, see Access MicroTx.

If the MicroTx coordinator is healthy, you will see the following response.

```
{
    "coordinators": {
        "xaCoordinator": "Healthy",
        "lraCoordinator": "Healthy",
        "tccCoordinator": "Healthy"
    },
    "version": "",
    "started": "2023-10-11T11:22:44.343082+05:30"
}
```

This indicates that you have successfully deployed MicroTx and the service is available to coordinate XA transactions.

5 Post-Installation Tasks

After installing Transaction Manager for Microservices (MicroTx), complete the following tasks to verify that the installation was successful and access MicroTx.

- Upgrade to 24.2
  MicroTx 24.2 provides additional features.
- 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.
- What's Next
  After installing MicroTx, you can integrate MicroTx client libraries with your application code or run sample applications to understand how MicroTx works.

5.1 Upgrade to 24.2

MicroTx 24.2 provides additional features.

Run one of the following steps only if you want to upgrade to MicroTx 24.2 to avail the latest features. For information about the new features, see What's New in MicroTx.

Topics

- Upgrade to the Latest Free Release
  MicroTx Free 24.2 provides additional features.
- Upgrade to the Latest Enterprise Edition in Kubernetes Cluster
  Run these steps to upgrade to the latest release of Transaction Manager for Microservices Enterprise Edition only if you use Oracle Database to store the transaction logs in the previous release. Run these steps on Kubernetes clusters in Kubernetes and OpenShift platforms.
- Upgrade to the Latest Enterprise Edition in Docker
  Run these steps to upgrade to the latest release of Transaction Manager for Microservices Enterprise Edition only if you use Oracle Database to store the transaction logs in the previous release. You can run these steps in the Docker environment.
- Upgrade to the Latest Enterprise Edition in Docker Swarm
  Run these steps to upgrade to the latest release of Transaction Manager for Microservices Enterprise Edition only if you use Oracle Database to store the transaction logs in the previous release. You can run these steps in the Docker Swarm environment.

5.1.1 Upgrade to the Latest Free Release

MicroTx Free 24.2 provides additional features.

To upgrade to MicroTx Free 24.2 from a previous release of MicroTx Free:

1. Download the files to install the latest release of MicroTx. See Download the Installation Bundle.

Ensure that you do not overwrite the files that you had downloaded for earlier versions of MicroTx.

The latest image of the transaction coordinator is available at `installation_directory/otmm-<version>/otmm/image/tmm-<version>.tgz`. In the next steps, you will use this file to update the existing transaction coordinator image.

2. Load the transaction coordinator image to the local repository, tag the image, and then push the image.
   - If you have installed MicroTx in a Kubernetes cluster, see Push Images to a Remote Docker Repository.
   - If you have installed MicroTx in Docker Swarm, see Push Image to a Docker Registry.
3. Update the YAML file for the transaction coordinator with the name of the latest image in the repository. If you want to use the latest features, complete the required tasks to use these features, and then update the YAML file with the property values.
   - If you have installed MicroTx in a Kubernetes cluster, see Configure the values.yaml File.
   - If you have installed MicroTx in Docker Swarm, see Configure the tcs-docker-swarm.yaml File.
4. If you have installed MicroTx in a Kubernetes cluster, run the following command to complete the upgrade.

   Syntax

   ```
   helm upgrade <release name> --namespace <namespace> <chart directory> --
   values <path_to_updated_values.yaml>
   ```

   The following sample command upgrades the MicroTx application named `tmm-app` in the `otmm` namespace.

   ```
   helm upgrade tmm-app --namespace otmm tmm --values tmm/values1.yaml
   ```

   Where,
   - `tmm-app` is the name of the MicroTx application.
   - `otmm` is the namespace in Kubernetes cluster, where you have installed MicroTx.
   - `installation_directory/otmm-<version>/otmm/helmcharts/tmm` is the folder that contains the `chart.yaml` file for MicroTx.
   - `installation_directory/otmm-<version>/otmm/helmcharts/tmm/values1.yaml` is the location of the application's updated manifest file in your local machine. This file contains the updated deployment configuration details for MicroTx.

5.1.2 Upgrade to the Latest Enterprise Edition in Kubernetes Cluster

Run these steps to upgrade to the latest release of Transaction Manager for Microservices Enterprise Edition only if you use Oracle Database to store the transaction logs in the previous release. Run these steps on Kubernetes clusters in Kubernetes and OpenShift platforms.

The MicroTx coordinator runs in the main container. As part of the upgrade process, MicroTx creates a Kubernetes init container. The Kubernetes init container is a specialized container that runs in a pod. The Kubernetes init container starts, completes the prerequisite steps for the upgrade, upgrades the MicroTx coordinator, and then terminates when it finishes the upgrade. It uses the MicroTx image that the main container also uses. See https://kubernetes.io/docs/concepts/workloads/pods/init-containers/.

Do not use the instructions in this section in the following scenarios:

- if you used etcd or internal memory to store the transaction logs in a previous release of Transaction Manager for Microservices Enterprise Edition.
- if you want to upgrade to the latest release of MicroTx Free.
- if you want to upgrade MicroTx in Docker or Docker Swarm. Docker does not support the init container functionality, which is available only in Kubernetes cluster. To upgrade in Docker environments, you must run the init process in an independent container. See Upgrade to the Latest Enterprise Edition in Docker or Upgrade to the Latest Enterprise Edition in Docker Swarm.

To upgrade to Transaction Manager for Microservices 24.2 Enterprise Edition:

1. Grant privileges to create and run stored procedures to the data store user. Perform this task only if you use Oracle Database as the data store. See Grant Privilege to Run Stored Procedures.
2. Ensure that you have the following required privileges to perform this task if you are not a database administrator.
   - EXECUTE
   - INSERT
   - SELECT
   - UPDATE
   - DELETE
   - CREATE
   - ALTER

   MicroTx uses these privileges to handle the upgrade.
3. Download the files to install the latest release of MicroTx. See Download the Installation Bundle.

   Ensure that you do not overwrite the installer files for earlier versions of MicroTx.

   The latest image of the transaction coordinator is available at `*installation_directory*/otmm-24.2/otmm/image/tmm-24.2.tgz`. In the next steps, you will use this file to update the existing transaction coordinator image.
4. Load the transaction coordinator image to the local repository, tag the image, and then push the image. See Push Images to a Remote Docker Repository.
5. Create a copy of the `values.yaml` file and name it as `values1.yaml`. Update the `values1.yaml` file with the name of the latest transaction coordinator image in the repository. If you want to use the latest features, complete the required tasks to use these features, and then update the YAML file with the property values. See Configure the values.yaml File.
6. Update the property values for the Kubernetes init container in the `tcs.yaml` file, which is located at `*installation_directory*/otmm-24.2/otmm/image/tcs.yaml`. Use this file as a

reference when you create your own Helm Chart to upgrade MicroTx. The following code snippet provides sample values. Provide the values based on your environment.

```
{{- if .Values.tmmConfiguration.storage.db }}
      {{- if eq .Values.tmmConfiguration.storage.type "db" }}
      initContainers:
        - name : {{ printf "init-%s" .Values.tmmConfiguration.tmmAppName }}
          image: {{ .Values.tmmImage.image }}
          securityContext:
            runAsUser: 1337
          volumeMounts:
            - name: config-volume
              mountPath: /etc/config
            {{-
if .Values.tmmConfiguration.storage.db.walletConfigMap.configMapName }}
            - name: wallet-volume
              mountPath: /etc/config/dbwallet
            {{- end }}
          env:
            - name: CONFIG_MAP_PATH
              value: /etc/config
            - name: INIT_CONTAINER_ENABLED
              value: "True"
            {{-
if .Values.tmmConfiguration.storage.db.credentialSecretName }}
            - name: STORAGE_DB_CREDENTIAL
              valueFrom:
                secretKeyRef:
                  key: secret
                  name:
{{ .Values.tmmConfiguration.storage.db.credentialSecretName }}
            {{- end }}
      {{- end }}
      {{- end }}
```

Parameters	Description

`securityContext:`
`            runAsUser: 1337`
	When you use a Kubernetes init container with an Istio service mesh, the Kubernetes init container cannot send any outgoing network calls as Istio does not support it. To enable the Kubernetes init container to send outgoing network calls, you must specify the security context. See https://istio.io/latest/docs/setup/additional-setup/cni/#compatibility-with-application-init-containers.

`INIT_CONTAINER_ENABLED`	Set this to `True` to enable the creation of a Kubernetes init container.

7. Run the following command to complete the upgrade.

Syntax

```
helm upgrade <release name> --namespace <namespace> <chart directory> --
values <path_to_updated_values.yaml>
```

The following sample command upgrades the MicroTx application named `tmm-app` in the `otmm` namespace.

```
helm upgrade tmm-app --namespace otmm tmm/ --values tmm/ee/values1.yaml
```

Where,

- `tmm-app` is the name of the MicroTx application.
- `otmm` is the namespace in Kubernetes cluster, where you have installed MicroTx.
- *`installation_directory`*`/otmm-24.2/otmm/helmcharts/tmm` is the folder that contains the `chart.yaml` file for MicroTx.
- *`installation_directory`*`/otmm-24.2/otmm/helmcharts/tmm/ee/values1.yaml` is the location of the `values1.yaml` file, the application's updated manifest file, in your local machine. This file contains the updated deployment configuration details for MicroTx.

5.1.3 Upgrade to the Latest Enterprise Edition in Docker

Run these steps to upgrade to the latest release of Transaction Manager for Microservices Enterprise Edition only if you use Oracle Database to store the transaction logs in the previous release. You can run these steps in the Docker environment.

The MicroTx coordinator runs in the main container. As part of the upgrade process, MicroTx creates an init process in an independent container. The init process starts, completes the prerequisite steps for the upgrade, upgrades the MicroTx coordinator, and then terminates when it finishes the upgrade. It uses the MicroTx image that the main container also uses.

Do not use the instructions in this section in the following scenarios:

- if you used etcd or internal memory to store the transaction logs in a previous release of Transaction Manager for Microservices Enterprise Edition.
- if you want to upgrade to the latest release of MicroTx Free.
- if you want to upgrade MicroTx in Kubernetes clusters. See Upgrade to the Latest Enterprise Edition in Kubernetes Cluster.

To upgrade to Transaction Manager for Microservices 24.2 Enterprise Edition:

1. Download the files for the latest release of MicroTx. See Download the Installation Bundle.

   Ensure that you do not overwrite the files that you had downloaded for earlier versions of MicroTx.

   The latest image of the transaction coordinator is available at *`installation_directory`*`/otmm-24.2/otmm/image/tmm-24.2.tgz`. In the next few steps, you will use this file to update the existing transaction coordinator image.

2. Load the MicroTx image to the local Docker repository. The MicroTx image is located at `installation_directory/otmm-<version>/image/tmm-<version>.tgz`.

   ```
   cd installation_directory/otmm-<version>/otmm
   docker load < image/tmm-<version>.tgz
   ```

   The following message is displayed when the image is loaded.

   ```
   Loaded image: tmm:<version>
   ```

3. Set `INIT_CONTAINER_ENABLED` to `True` in the `tcs.yaml` file to enable the creation of an init process. The `tcs.yaml` file is located in the *`installation_directory`*`/otmm-24.2/otmm/image` folder.
4. Run the following command to initiate an init process in a Docker container to upgrade MicroTx.

   Syntax

   ```
   docker container run --name init-otmm-app \
   -v "$(pwd)":/app/config \
   -w /app \
   -p 9000:9000/tcp \
   --env CONFIG_FILE=/app/config/tcs.yaml \
   --network=host \
   -e STORAGE_DB_CREDENTIAL='{"password":"<dbUserpassword>",
   "username":"<DBuserName>"}' \
   -d tmm:24.2
   ```

   Where,

   - `--name init-otmm-app` is the name of the MicroTx application that you want to upgrade.
   - `-v "$(pwd)":/app/config` mounts the current directory into container at the `/app/config` path.
   - `-w /app` specifies the working directory as `/app`. In this sample command, `/app` remains the default working directory as the volume is also mounted on this directory based on the value provided for the `-v` flag.
   - `--env CONFIG_FILE=config/tcs.yaml` specifies the location of the `tcs.yaml` file, which contains the coordinator configuration details.
   - `-e STORAGE_DB_CREDENTIAL` provides the user name and password to access Oracle Database.
   - `tmm:<version>` is the MicroTx Docker image that you have loaded to the local Docker repository.
5. Remove the independent container, in which the init process ran, from the list of docker containers in Docker environment. The init process terminates automatically after it finishes the upgrade. Run the following command to remove the independent container.

   ```
   docker container rm <container_ID_of_init_otmm_app>
   ```

6. Set `INIT_CONTAINER_ENABLED` to `False` in the `tcs.yaml` file.
7. Run the following command to start running MicroTx in the Docker container.

   Syntax

   The following sample command runs the MicroTx application named `otmm-app`.

   ```
   docker container run --name otmm-app \
   -v "$(pwd)":/app/config \
   -w /app \
   -p 9000:9000/tcp \
   --env CONFIG_FILE=/app/config/tcs.yaml \
   --network=host \
   ```

```
-e STORAGE_DB_CREDENTIAL='{"password":"<dbUserpassword>",
"username":"<DBuserName>"}' \
-d tmm:24.2
```

Where,

- `--name otmm-app` is the name of the MicroTx application that you want to create.
- `-v "$(pwd)":/app/config` mounts the current directory into container at the `/app/config` path.
- `-w /app` specifies the working directory as `/app`. In this sample command, `/app` remains the default working directory as the volume is also mounted on this directory based on the value provided for the `-v` flag.
- `--env CONFIG_FILE=config/tcs.yaml` specifies the location of the `tcs.yaml` file, which contains the coordinator configuration details.
- `tmm:<version>` is the MicroTx Docker image that you have loaded to the local Docker repository.

5.1.4 Upgrade to the Latest Enterprise Edition in Docker Swarm

Run these steps to upgrade to the latest release of Transaction Manager for Microservices Enterprise Edition only if you use Oracle Database to store the transaction logs in the previous release. You can run these steps in the Docker Swarm environment.

The MicroTx coordinator runs in the main container. As part of the upgrade process, MicroTx creates an init process in an independent container. The init process starts, completes the prerequisite steps for the upgrade, upgrades the MicroTx coordinator, and then terminates when it finishes the upgrade. It uses the MicroTx image that the main container also uses.

Do not use the instructions in this section in the following scenarios:

- if you used etcd or internal memory to store the transaction logs in a previous release of Transaction Manager for Microservices Enterprise Edition.
- if you want to upgrade to the latest release of MicroTx Free.
- if you want to upgrade MicroTx in Docker. See Upgrade to the Latest Enterprise Edition in Docker.
- if you want to upgrade MicroTx in Kubernetes clusters. See Upgrade to the Latest Enterprise Edition in Kubernetes Cluster.

To upgrade to Transaction Manager for Microservices 24.2 Enterprise Edition:

1. Download the files for the latest release of MicroTx. See Download the Installation Bundle.

   Ensure that you do not overwrite the files that you had downloaded for earlier versions of MicroTx.

   The latest image of the transaction coordinator is available at *`installation_directory`*`/otmm-24.2/otmm/image/tmm-24.2.tgz`. In the next few steps, you will use this file to update the existing transaction coordinator image.
2. Load the transaction coordinator image to the local repository, tag the image, and then push the image. See Push Image to a Docker Registry.
3. Set `INIT_CONTAINER_ENABLED` to `True` in the `tcs.yaml` file to enable the creation of an init process. The `tcs.yaml` file is located in the *`installation_directory`*`/otmm-24.2/otmm/image` folder.

4. Run the following command to initiate an init process in a Docker container to upgrade MicroTx.

   Syntax

   ```
   docker container run --name init-otmm-app \
   -v "$(pwd)":/app/config \
   -w /app \
   -p 9000:9000/tcp \
   --env CONFIG_FILE=/app/config/tcs.yaml \
   --network=host \
   -e STORAGE_DB_CREDENTIAL='{"password":"<dbUserpassword>",
   "username":"<DBuserName>"}' \
   -d tmm:24.2
   ```

   Where,

   - `--name init-otmm-app` is the name of the MicroTx application that you want to upgrade.
   - `-v "$(pwd)":/app/config` mounts the current directory into container at the `/app/config` path.
   - `-w /app` specifies the working directory as `/app`. In this sample command, `/app` remains the default working directory as the volume is also mounted on this directory based on the value provided for the `-v` flag.
   - `--env CONFIG_FILE=config/tcs.yaml` specifies the location of the `tcs.yaml` file, which contains the coordinator configuration details.
   - `-e STORAGE_DB_CREDENTIAL` provides the user name and password to access Oracle Database.
   - `tmm:<version>` is the MicroTx Docker image that you have loaded to the local Docker repository.

5. Remove the independent container, in which the init process ran, from the list of docker containers in Docker environment. The init process terminates automatically after it finishes the upgrade. Run the following command to remove the independent container.

   ```
   docker container rm <container_ID_of_init_otmm_app>
   ```

6. Update the `tmm-stack-compose.yaml` file to provide the name of the latest MicroTx coordinator image.

7. Perform one of the following tasks to complete the upgrade process depending on your Docker Swarm environment.

   - If a Docker stack is not deployed or if you want to create a new stack, run the following command to deploy a new Docker stack.
     Command Syntax

     ```
     docker stack deploy -c tmm-stack-compose.yaml <name_of_stack>
     ```

   - If a Docker stack is already deployed and running, run the following command to update the stack.

Command Syntax

```
docker service update --image {latest image of MicroTx coordinator}
{name of the MicroTx
    service}
```

Sample Command

```
docker service update --image 198.51.100.1:5000/tmm:1679 tmmdemo_otmm-
tcs
```

5.2 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........   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.

5.3 What's Next

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

The MicroTx 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.

6 Develop Applications with XA

To use Transaction Manager for Microservices (MicroTx) 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 libraries.

The MicroTx library is available for Java, Node.js, ORDS, Tuxedo, and WebLogic Server apps. See Supported Languages and Frameworks.

1. Before you begin, ensure that you have installed the MicroTx transaction coordinator and you can access it. See Access MicroTx.
2. Include the MicroTx client libraries in your microservice implementation.
3. Use CDI annotations or MicroTx client libraries APIs to register the required interceptors and callbacks.
4. Use CDI annotations or MicroTx client library APIs in participant microservices to obtain the connection to their XA compliant resource manager.
5. Use MicroTx client libraries API to delineate transaction boundaries indicating an XA transaction has started, and then commit or roll back the transaction.

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

Task	Description	More Information
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
Provide configuration information for the MicroTx 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 MicroTx 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 REST 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

- 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 varies 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
  Provide configuration information for the MicroTx library properties for JAX-RS, Spring REST, 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 REST Apps with XA
  Use the MicroTx library with your Spring REST applications.
- Develop JAX-RS Apps with XA
  Use the MicroTx library with your JAX-RS applications.
- Configure Java Apps to Leverage Transactional Event Queues
  Integrate Oracle MicroTx 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.

6.1 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 varies 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 server the restarts after a failure.

6.1.1 Supported Resource Managers

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

In XA transactions, the MicroTx 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 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 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 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 library does not interact with the application database in Saga and TCC transaction protocols.

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

The MicroTx 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 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 library accesses the `XAResource` object to perform various XA operations on the resource manager.

6.1.3 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, 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 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 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.

6.1.4 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 creates a new branch for each department. In all MicroTx 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 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 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 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.

6.1.5 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 client libraries for Java applications. JPA or Hibernate applications support only XA-compliant resource managers.

6.1.6 About Dynamic Recovery for XA Transactions

The transaction coordinator server resumes the transactions that were in progress when server the 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 and 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 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.

6.2 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 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.

6.3 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.

6.4 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 to begin an XA transaction, by default this transaction is handled as a global transaction.
- Subscribe to Receive XA Transaction Notifications
  You can register your transaction initiator and participant services to receive notifications. MicroTx notifies the registered services when the following events occur: before the prepare phase and when MicroTx successfully commits or rolls back a transaction.

6.4.1 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. If you specify this value for a participant application, it is ignored.

To set transaction timeout for requests sent from participants services:

1. For the `txMaxTimeout` parameter in the `values.yaml` file of the MicroTx, specify the *maximum* amount of time, in milliseconds, for which a transaction remains active. The default value is 60000 ms.

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

2. For the `ORACLE_TMM_TRANSACTION_TIMEOUT` parameter in the `tmm.properties` file of the transaction initiator service, 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.

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

6.4.2 About Global and Local Transactions

When a transaction initiator service makes a call to MicroTx 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 client library manages the local transactions. By handling a transaction locally, without using the MicroTx coordinator, you can experience better performance. Local transactions save time and increase throughput.

- 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.

6.4.2.1 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.

6.4.2.2 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.

6.4.3 Subscribe to Receive XA Transaction Notifications

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

The MicroTx 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 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 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 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 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 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.

6.5 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.

6.6 Configure Library Properties

Provide configuration information for the MicroTx library properties for JAX-RS, Spring REST, 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. 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.

- `oracle.tmm.TcsUrl` for JAX-RS apps or `spring.microtx.coordinator-url` for Spring REST apps: Enter the URL to access the MicroTx application. See Access MicroTx. 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 library to MicroTx. 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 participant service. MicroTx 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 participant service remotely. For example, `8081`.

  You must specify this value for the transaction participant applications. You don't have to specify this value for the transaction initiator application.
- `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 for both initiator and participant applications.
- `oracle.tmm.PropagateTraceHeaders` for JAX-RS apps or `spring.microtx.propagation-active` for Spring REST 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. 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 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 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 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 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.

6.7 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/javaee/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.

6.8 Develop Spring REST Apps with XA

Use the MicroTx library with your Spring REST applications.

- Configure Library Properties for Spring REST Apps
  Provide configuration information for the MicroTx library properties for every Spring REST application. The property values that you must provide would vary depending on whether the application is a participant or an initiator.
- Configure Spring REST 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 REST 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 library to configure your Spring REST participant application.

6.8.1 Configure Library Properties for Spring REST Apps

Provide configuration information for the MicroTx library properties for every Spring REST 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 library in the `application.yaml` file.

- `spring.microtx.coordinator-url`: Enter the URL to access the MicroTx coordinator. See Access MicroTx. 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 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 participant service remotely. For example, `8081`.

  You must specify this value for the transaction participant applications. You don't have to specify this value for the transaction initiator application.

  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 library to MicroTx. 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 for both initiator and participant applications.
- `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 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 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_LLR_SUPPORT = false
export SPRING_MICROTX_XA_LLR_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.

6.8.2 Configure Spring REST 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 based on Spring REST as a transaction initiator:

1. Specify property values for the MicroTx library. See Configure Library Properties for Spring REST Apps.
2. Include only one of the following MicroTx library files as a maven dependency in the Spring Boot 3.x application's `pom.xml` file. The following sample code is for the 24.2 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.
   - For JDBC applications, use the `microtx-spring-boot-starter` file.

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter</artifactId>
          <version>24.2</version>
     </dependency>
     ```

   - For Java Apps that use Hibernate as JPA provider, use the `microtx-spring-boot-starter-hibernate` library file.

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
          <version>24.2</version>
     </dependency>
     ```

   - For Java Apps that use EclipseLink as JPA provider, use the `microtx-spring-boot-starter-eclipselink` library file.

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
          <version>24.2</version>
     </dependency>
     ```

3. 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)
        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 REST apps, see https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html.

   c. Skip to step 8 as you don't have to specify the transaction boundaries using `begin()`, `commit()`, and `rollback()` when you use `@Transactional`.

4. Import the `com.oracle.microtx.xa.rm.MicroTxUserTransactionService` package.

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

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

   ```
   @Autowired
   MicroTxUserTransactionService microTxUserTransactionService;
   ```

6. 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.
     ```

7. 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();
   ```

```
        }
    }
```

8. Use a WebClient or the REST template 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 object that you create.

   -
     ```
     @Autowired
     @Qualifier("MicroTxXaWebClientBuilder")
     WebClient.Builder webClientBuilder;
     ```

   - Inject the Spring Boot REST template provided by MicroTx as shown in the following sample code.

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

9. Save your changes.

The sample XA application code for transaction initiator service is located at `*installation_directory*\otmm-*RELEASE*\samples\xa\java\teller-as-participant-spring`. This provides an example of how you can use the MicroTx 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 REST App as Transaction Participant .

6.8.3 Configure Spring REST 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 library to configure your Spring REST participant application.

Note:

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

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

- Configure Spring REST App with a Non-XA and Non-JDBC Resource
  Use the information provided in this section to configure your Spring REST participant applications when you use a resource that does not support XA and JDBC.
- Configure JPA-Based Spring REST 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 REST applications which participates in an XA transaction when you use an XA-compliant resource manager.

6.8.3.1 Configure Spring REST Apps with an XA-Compliant Resource Manager

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

1. Configure property values for the MicroTx 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 REST Apps.

2. Include only one of the following MicroTx library files as a maven dependency in the Spring Boot 3.x application's `pom.xml` file. The following sample code is for the 24.2 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

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

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter</artifactId>
          <version>24.2</version>
     </dependency>
     ```

   - For Java Apps that use Hibernate as JPA provider, use the `microtx-spring-boot-starter-hibernate` library file.

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
          <version>24.2</version>
     </dependency>
     ```

- For Java Apps that use EclipseLink as JPA provider, use the `microtx-spring-boot-starter-eclipselink` library file.

  ```
  <dependency>
       <groupId>com.oracle.microtx</groupId>
       <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
       <version>24.2</version>
  </dependency>
  ```

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

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

4. Initialize an `XADatasource` object.

   The MicroTx 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 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 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 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 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 client library. The following code in the participant application injects the `connection` object that is created by the MicroTx client library.

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

8. 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 client library.

9. Save the changes.

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

6.8.3.2 Configure Spring REST App with Multiple XA-Compliant Resource Managers

Use the information provided in this section to configure your Spring REST 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 only one of the following MicroTx library files as a maven dependency in the Spring Boot 3.x application's `pom.xml` file. The following sample code is for the 24.2 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

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

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter</artifactId>
          <version>24.2</version>
     </dependency>
     ```

   - For Java Apps that use Hibernate as JPA provider, use the `microtx-spring-boot-starter-hibernate` library file.

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

```
        <version>24.2</version>
    </dependency>
```

- For Java Apps that use EclipseLink as JPA provider, use the `microtx-spring-boot-starter-eclipselink` library file.

```
    <dependency>
         <groupId>com.oracle.microtx</groupId>
         <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
         <version>24.2</version>
    </dependency>
```

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

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

3. 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.

4. Enter only *one* of the following MicroTx 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();
  ```

5. 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 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.

6. In the transaction participant function or block, specify the `XADatasource` object which is used by the MicroTx 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 client library uses the `XADatasource` object to create database connections.

   Repeat this step for every resource manager.

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(dataSource, creditDataSource)
   ```

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

   The MicroTx 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 library also provides a SQL connection object to the application code to execute DML using dependency injection.

Repeat this step for every resource manager.

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

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 client library.

Repeat this step for every resource manager.

10. Save the changes.

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

6.8.3.3 Configure Spring REST App with a Non-XA JDBC Resource

Use the information provided in this section to configure your Spring REST 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 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 REST Apps.

2. If you are using a multiple resource managers with your application, complete the following steps to configure property values for the MicroTx 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 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 only one of the following MicroTx library files as a maven dependency in the Spring Boot 3.x application's `pom.xml` file. The following sample code is for the 24.2 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

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

```
<dependency>
     <groupId>com.oracle.microtx</groupId>
     <artifactId>microtx-spring-boot-starter</artifactId>
     <version>24.2</version>
</dependency>
```

   - For Java Apps that use Hibernate as JPA provider, use the `microtx-spring-boot-starter-hibernate` library file.

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

```
        <version>24.2</version>
    </dependency>
```

- For Java Apps that use EclipseLink as JPA provider, use the `microtx-spring-boot-starter-eclipselink` library file.

  ```
  <dependency>
       <groupId>com.oracle.microtx</groupId>
       <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
       <version>24.2</version>
  </dependency>
  ```

4. Enable session affinity. See Enable Session Affinity.
5. Import the `com.oracle.microtx.common.MicroTxNonXAConfig` package.

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

6. Initialize a `Datasource` object.

   The MicroTx 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.

7. In the transaction participant function or block, specify the `DataSource` object which is used by the MicroTx 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 library uses the `DataSource` object to create database connections.

8. In the transaction participant function or block, add the following line of code only once after you have initialized the `Datasource` object. The MicroTx library uses this object to start a database transaction. The MicroTx 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.

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

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

10. 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 library.

11. Save the changes.

6.8.3.4 Configure Spring REST App with a Non-XA and Non-JDBC Resource

Use the information provided in this section to configure your Spring REST 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 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 REST Apps.

2. Include only one of the following MicroTx library files as a maven dependency in the Spring Boot 3.x application's `pom.xml` file. The following sample code is for the 24.2 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

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

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter</artifactId>
          <version>24.2</version>
     </dependency>
     ```

   - For Java Apps that use Hibernate as JPA provider, use the `microtx-spring-boot-starter-hibernate` library file.

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
          <version>24.2</version>
     </dependency>
     ```

   - For Java Apps that use EclipseLink as JPA provider, use the `microtx-spring-boot-starter-eclipselink` library file.

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
          <version>24.2</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 library files, and then produce a non-XA resource. Annotate the non-XA resource that you create with `@NonXa` annotation. The MicroTx 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 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();
       }
   }
   ```

6. Save the changes.

6.8.3.5 Configure JPA-Based Spring REST 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 REST 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 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 REST Apps.

2. Include only one of the following MicroTx library files as a maven dependency in the Spring Boot 3.x application's `pom.xml` file. The following sample code is for the 24.2 release. Provide the correct version, based on the release version that you want to use. Spring Boot 3.x applications work with Java 17.

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

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter</artifactId>
          <version>24.2</version>
     </dependency>
     ```

   - For Java Apps that use Hibernate as JPA provider, use the `microtx-spring-boot-starter-hibernate` library file.

     ```
     <dependency>
          <groupId>com.oracle.microtx</groupId>
          <artifactId>microtx-spring-boot-starter-hibernate</artifactId>
          <version>24.2</version>
     </dependency>
     ```

- For Java Apps that use EclipseLink as JPA provider, use the `microtx-spring-boot-starter-eclipselink` library file.

  ```
  <dependency>
       <groupId>com.oracle.microtx</groupId>
       <artifactId>microtx-spring-boot-starter-eclipselink</artifactId>
       <version>24.2</version>
  </dependency>
  ```

3. 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 REST 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 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
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 EcpliseLink apps using Spring REST 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;
    }
}
```

4. 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 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 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).
5. Insert the following lines in the code of the participant service so that the application uses the connection passed by the MicroTx client library. The following code in the participant application injects the `connection` object that is created by the MicroTx 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.

6. In your application code, inject the entity manager object that you have passed to the MicroTx 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.
7. Save the changes.

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

6.9 Develop JAX-RS Apps with XA

Use the MicroTx library with your JAX-RS applications.

- About MicroTx library for JAX-RS Apps
  Integrate the MicroTx 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 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.

6.9.1 About MicroTx library for JAX-RS Apps

Integrate the MicroTx library with your JAX-RS applications.

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

- Enlists the participant service with the Transaction Coordinator in the transaction.
- Injects an `XADataSource` 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 `XADataSource` object.
  The MicroTx 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.

6.9.2 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 library. See Configure Library Properties.
2. Include the MicroTx Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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>24.2</version>
  </dependency>
  ```

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

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

4. 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();
   ```

5. 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.
     ```

6. 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.

7. 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 an XA transaction initiator application is available in the `xa/java/teller` folder in the `microtx-samples` GitHub repository. This provides an example of how you can use MicroTx Java libraries with the business logic of your Java 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. Use this as a reference while integrating the MicroTx libraries with 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 JAX-RS App as Transaction Participant.

6.9.3 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 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.

6.9.3.1 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 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.

2. Include the MicroTx Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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>24.2</version>
     </dependency>
     ```

3. Initialize an `XADatasource` object.

   The MicroTx 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 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 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 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 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 client library. The following code in the participant application injects the `connection` object that is created by the MicroTx 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 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 XA resource callback which
              // coordinates with the transaction coordinator
              register(XAResourceCallbacks.class);
              // Register the filters for the MicroTx 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 connection factory object, so that MicroTx 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.

6.9.3.2 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 Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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>24.2</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 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.

3. Enter only *one* of the following MicroTx 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 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 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 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 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 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 XA resource callback which
        // coordinates with the transaction coordinator
        register(XAResourceCallbacks.class);
        // Register the filters for the MicroTx 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 client library. The following code in the participant application injects the `connection` object that is created by the MicroTx 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 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.

6.9.3.3 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 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 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 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 Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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>24.2</version>
     </dependency>
     ```

4. Enable session affinity. See Enable Session Affinity.
5. Initialize a `Datasource` object.

   The MicroTx 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 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 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 library uses this object to start a database transaction. The MicroTx 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 library. The following code in the participant application injects the `connection` object that is created by the MicroTx library.

   ```
   @Inject @TrmNonXASQLConnection 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 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.

6.9.3.4 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 library. See Configure Library Properties.

   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 Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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>24.2</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 library files, and then produce a non-XA resource. Annotate the non-XA resource that you create with `@NonXa` annotation. The MicroTx 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 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.

6.9.4 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 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 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.

6.9.4.1 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 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.

2. Include the MicroTx Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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>24.2</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.mtm.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 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 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 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 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 XA resource callback which
           // coordinates with the transaction coordinator
           register(XAResourceCallbacks.class);
           // Register the filters for the MicroTx 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 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 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 client library. The following code in the participant application injects the `connection` object that is created by the MicroTx 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 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.

6.10 Configure Java Apps to Leverage Transactional Event Queues

Integrate Oracle MicroTx 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 library files with the publisher.

Application developers use a custom annotation, `MicroTxXATopicSession`, provided by the MicroTx library to inject a TEQ session and use it in their application code. The MicroTx 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 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 library with your Spring REST participant application to leverage the Oracle Transactional Event Queues (TEQ) feature.

6.10.1 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>');
```

6.10.2 Configure JAX-RS Participant App to Leverage Transactional Event Queues

Use the information provided in this section to integrate Oracle MicroTx 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 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 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.

2. Import the `oracle.tmm.jta.common.MicroTxXATopicSession` package.

   ```
   import oracle.tmm.jta.common.MicroTxXATopicSession;
   ```

3. Include the MicroTx Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</version>
   </dependency>
   ```

4. Initialize an `XADatasource` object.

   The MicroTx 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 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 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 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 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 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 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 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 client library.

10. Save the changes.

Source code of a sample JAX-RS transaction participant application which uses the MicroTx 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.

6.10.3 Configure Spring REST Participant App to Leverage Transactional Event Queues

Use the information provided in this section to integrate Oracle MicroTx 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 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: deptxadatasource
```

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 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 REST Apps.

2. Include the MicroTx Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</version>
</dependency>
```

3. Import the `jakarta.jms.XATopicSession` package.

   ```
   import jakarta.jms.XATopicSession;
   ```

4. Initialize an `XADatasource` object.

   The MicroTx 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 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 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 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 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 client library. The following code in the participant

application injects the `microTxXATopicSession` object that is created by the MicroTx 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 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 library also provides a SQL connection object to the application code to execute DML using dependency injection.

8. When you inject `MicroTxXATopicSession`, MicroTx 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 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 client library.

10. Save the changes.

Source code of a sample Spring REST transaction participant application which uses the MicroTx XA library functions to leverage TEQ is available in the `xa/java/jms/department-spring-teq` folder in the `microtx-samples` GitHub repository. You can use this as a reference while integrating the MicroTx libraries with your application.

6.11 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 client library.

6.11.1 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 library for Node.js as a dependency in the `package.json` file. The library file is located in the *installation_directory*/otmm-*RELEASE*/otmm/nodejs folder.

   ```
   "dependencies": {
       "tmmlib-node": "file:oracle-microtx-1.0.0.tgz"
     }
   ```

2. Configure the property values for the MicroTx library. See Configure Library Properties. To enable logging for Node.js applications, you must set additional properties. See Enable Logs for MicroTx Node.js Library.
3. Configure the MicroTx 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();
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 .

6.11.2 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 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.

6.11.2.1 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 library for Node.js as a dependency in the `package.json` file. The library file is located in the *`installation_directory`*`/otmm-`*`RELEASE`*`/otmm/nodejs` folder.

   ```
   "dependencies": {
       "tmmlib-node": "file:oracle-microtx-1.0.0.tgz"
     }
   ```

2. Configure the property values for the MicroTx library. See Configure Library Properties. To enable logging for Node.js applications, you must set additional properties. See Enable Logs for MicroTx 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 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 libraries.

   ```
   import {Request, Response, Router} from 'express';
   import {XATransactionMethod, XAConfig, XADataSource, TrmXAResource} from
   "tmmlib-node/xa/xa";
   import {TrmConfig} from "tmmlib-node/util/trmutils";
   import {asyncHandler} from "tmmlib-node/util/asynchandler";
   ```

5. If you are using Oracle Database as the resource manager, additionally import the following library.

   ```
   import {OracleXADataSource} from "tmmlib-node/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",
   connectString : "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 infect 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 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();
    }
}
```

6.11.2.2 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 library for Node.js as a dependency in the `package.json` file. The library file is located in the *`installation_directory`*`/otmm-`*`RELEASE`*`/otmm/nodejs` folder.

   ```
   "dependencies": {
       "tmmlib-node": "file:oracle-microtx-1.0.0.tgz"
     }
   ```

2. Configure the property values for the MicroTx library. See Configure Library Properties. To enable logging for Node.js applications, you must set additional properties. See Enable Logs for MicroTx 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 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 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 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 library.

   ```
   const nonxaResource: NonXAResource = new MongoNonXAResource();
   ```

8. Use the `TrmNonXAResource.init()` function to specify the `NonXAResource` object that the MicroTx library uses.

   ```
   TrmNonXAResource.init(nonxaResource)
   ```

9. Save the changes.

6.12 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.

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, an Oracle *RAD* stack deployed in a Kubernetes cluster, or a 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 PL/SQL Library
  The MicroTx 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.
- Configure PL/SQL Library Properties for ORDS Apps
  Provide configuration information for the MicroTx 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 PL/SQL library.
- Configure ORDS App as Transaction Initiator
- Configure ORDS App as Transaction Participant
- MicroTx PL/SQL Library Functions
  The MicroTx PL/SQL library provides the following functions to specify the transaction boundaries.

6.12.1 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 runs or it can run in any other environment.
- Ensure that there is network access or connectivity between MicroTx 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 library makes an outbound REST call to the MicroTx 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.

  – *`#PRINCIPAL_NAME`*: Enter the name of the principal user of APEX.

6.12.2 Run MicroTx PL/SQL Library

The MicroTx 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.

ORDS exposes REST APIs that are defined in Oracle Database using PL/SQL and APEX libraries. Integrate the MicroTx 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 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 PL/SQL library.

The exceptions thrown by the MicroTx PL/SQL library are defined in the `tmm_exceptions` package.

6.12.3 Configure PL/SQL Library Properties for ORDS Apps

Provide configuration information for the MicroTx 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 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 coordinator. See Access MicroTx. 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 participant service. MicroTx 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 participant service remotely. For example, `8081`.

  You must specify this value for the transaction participant applications. You don't have to specify this value for the transaction initiator application.

  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 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 PL/SQL library in the `microtx_log_data` table. Default value is `true`.

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'
```

6.12.4 Privileges

Set the privileges required to run the MicroTx PL/SQL library.

The following privileges are required to run the MicroTx PL/SQL library:

- `CREATE SESSION`
- `CREATE TABLE`
- `CREATE PROCEDURE`
- `CREATE SEQUENCE`
- `CREATE TYPE`

To perform XA operations using the MicroTx 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 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*.

6.12.5 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 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 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.
            l_microTxTransaction := tmm_begin_tx(l_forwardHeaders);

            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 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.

6.12.6 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 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 
than 0)
                            //or not (value of l_tmmReturn.proceed is 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 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 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.

6.12.7 MicroTx PL/SQL Library Functions

The MicroTx 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 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.

6.12.7.1 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 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 `tmmConfiguration.authorization.authTokenPropagationEnabled` to `true` in the `values.yaml` file for the coordinator, your application must forward the authorization token. If you have set `tmmConfiguration.transactionToken.transactionTokenEnabled` to `true` in the `values.yaml` 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 `xa-transaction-timeout` in the `microtx_config` table.

Return Values

Parameter	Description
`MicroTxTransaction`	Returns the transaction object `MicroTxTransaction`. 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.

6.12.7.2 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 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 `tmmConfiguration.authorization.authTokenPropagationEnabled` to `true` in the `values.yaml` file for the coordinator, your application must forward the authorization token. If you have set `tmmConfiguration.transactionToken.transactionTokenEnabled` to `true` in the `values.yaml` 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.

6.12.7.3 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.
`forwardHeaders`	Optional. The application forwards the specified headers to the MicroTx 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 `tmmConfiguration.authorization.authTokenPropagationEnabled` to `true` in the `values.yaml` file for the coordinator, your application must forward the authorization token. If you have set `tmmConfiguration.transactionToken.transactionTokenEnabled` to `true` in the `values.yaml` 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.

6.12.7.4 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 `MicroTxTransaction` object that the `TMM_BEGIN_TX` function returns.

Parameter	Description
`forwardHeaders`	Optional. The application forwards the specified headers to the MicroTx 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 `tmmConfiguration.authorization.authTokenPropagationEnabled` to `true` in the `values.yaml` file for the coordinator, your application must forward the authorization token. If you have set `tmmConfiguration.transactionToken.transactionTokenEnabled` to `true` in the `values.yaml` 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.

6.12.7.5 TmmStart Function

Enlists the transaction participant service with the MicroTx 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, `http://tmm-app:8080/ords/otmm/accounts`. Where, `/accounts` 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 `tmmConfiguration.authorization.authTokenPropagationEnabled` to `true` in the `values.yaml` file for the coordinator, your application must forward the authorization token.
`tmmTxToken`	Mandatory. If you have set `tmmConfiguration.transactionToken.transactionTokenEnabled` to `true` in the `values.yaml` 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` value	**Indicates that...**
0	the `TmmStart` 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 `TmmStart` 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 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.

6.12.7.6 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);
```

7

Develop Applications with Saga

The Transaction Manager for Microservices (MicroTx) 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.

Before you begin, ensure that you have installed MicroTx and access it.

Develop, test, and deploy your microservices independently. To use MicroTx 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 libraries.

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

Task	**Description**	**More Information**
Provide configuration information for the MicroTx 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
Integrate MicroTx 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 using the library files in your application, install the application.	Deploy Your Application

- 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.
- Configure Library Properties for Node.js Apps
  Provide configuration information for the MicroTx library properties. You must perform this step for all the Node.js applications which participate or initiate the transaction.

- Develop Node.js Apps with Saga
  The MicroTx 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.

7.1 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 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.
2. MicroTx responds to the transaction initiator and returns `Oracle_Tmm_Tx_Token` in the response header.
   The MicroTx 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 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 library passes `Oracle_Tmm_Tx_Token` in the request header for all the subsequent calls.
4. MicroTx 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.

7.2 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 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 library file.

- Configure MicroTx Library Properties for Java Apps
  Provide configuration information for the MicroTx 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 library.
- Develop Helidon Applications with Saga
  Helidon provides implementation for the Saga client specifications.
- Develop Spring REST-based Applications with Saga
  For Spring REST-based applications that use the Saga transaction protocol, you must use the Java library file provided by MicroTx.
- Develop Micronaut Apps with Saga
  You can integrate the MicroTx library files with Micronaut applications that use the Saga transaction protocol.
- Develop Java Apps that Use Saga and Lock-Free Reservation
  MicroTx Saga libraries for Spring REST and Micronaut applications support lock-free reservation.

7.2.1 Configure MicroTx Library Properties for Java Apps

Provide configuration information for the MicroTx 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 library.

Do not perform this step for JAX-RS or Helidon apps as these apps do not use the MicroTx 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 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 Saga coordinator. This permits the MicroTx library to communicate and coordinate transactions with the coordinator. To get the URL, append `/lra-coordinator` to the URL to access MicroTx. For example, `https://192.0.2.1:443/api/v1/lra-coordinator`. For information to identify the URL to access MicroTx, see Access MicroTx. 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 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 client library. MicroTx 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 participant service remotely. For example, `8081`.

  You must specify this value for the transaction participant applications. You don't have to specify this value for the transaction initiator application.
- `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 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 coordinator use the lock-free reservation feature provided in Oracle Database 23ai. When you set this value to `true`, MicroTx 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 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 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 are in the same Kubernetes cluster, otherwise use the HTTPS protocol.

7.2.2 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 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.

7.2.3 Develop Spring REST-based Applications with Saga

For Spring REST-based applications that use the Saga transaction protocol, you must use the Java library file provided by MicroTx.

To include the MicroTx library files and property values for Spring REST-based applications:

1. Include the MicroTx Java library file as a maven dependency in the Spring REST-based 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 library properties in the `application.properties` file of your application. This enables the custom library to establish communication with the MicroTx Saga coordinator, participate in Saga transactions, and propagate the relevant headers for the coordinated transactions. See Configure MicroTx Library Properties for Java Apps.

3. Import the `com.oracle.microtx.springboot.lra.annotation.*` package.

   ```
   import com.oracle.microtx.springboot.lra.annotation.*
   ```

4. Optional. Perform the following steps only if you want to execute the methods in the Spring-based REST 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 library. The following example shows how you can create a bean named `taskExecutorForTripBooking`, and then and then set `MicroTxTaskDecorator`, the MicroTx 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
       }
   ```

7.2.4 Develop Micronaut Apps with Saga

You can integrate the MicroTx library files with Micronaut applications that use the Saga transaction protocol.

To integrate the MicroTx library files with your Micronaut applications:

1. Include the MicroTx Java library file as a maven dependency in the Micronaut application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</version>
   </dependency>
   ```

2. Specify values for the MicroTx library properties in the `application.properties` file of your application. This enables the custom library to establish communication with the MicroTx Saga coordinator, participate in Saga transactions, and propagate the relevant headers for the coordinated transactions. See Configure MicroTx Library Properties for Java Apps.
3. Import the `com.oracle.microtx.lra.annotation.*` package.

   ```
   import com.oracle.microtx.lra.annotation.*
   ```

7.2.5 Develop Java Apps that Use Saga and Lock-Free Reservation

MicroTx Saga libraries for Spring REST and Micronaut applications support lock-free reservation.

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 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 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 Library Properties for Java Apps.
2. Import the `com.oracle.microtx.lra.lockfree` package.

   ```
   import com.oracle.microtx.lra.lockfree;
   ```

3. Leverage the MicroTx 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 library. The sample code 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.

7.3 Configure Library Properties for Node.js Apps

Provide configuration information for the MicroTx 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 library.

- `oracle.tmm.TcsUrl`: Enter the URL to access the MicroTx Saga coordinator. This permits the MicroTx library to communicate and coordinate transactions with the coordinator. To get the URL, append `/lra-coordinator` to the URL to access MicroTx. For example, `https://192.0.2.1:443/api/v1/lra-coordinator`. For information to identify the URL to access MicroTx, see Access MicroTx. 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 client library. MicroTx 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 participant service remotely. For example, `8081`.

You must specify this value for the transaction participant applications. You don't have to specify this value for the transaction initiator application.

- `oracle.tmm.PropagateTraceHeaders`: Set this to `true` when you want to trace the transaction from end-to-end. This propagates the Saga context headers 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 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.

7.4 Develop Node.js Apps with Saga

The MicroTx 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.

1. Add the MicroTx library for Node.js as a dependency in the `package.json` file. The library file is located in the `*installation_directory*/otmm-*RELEASE*/otmm/nodejs` folder.

   ```
   "dependencies": {
       "tmmlib-node": "file:oracle-microtx-1.0.0.tgz"
     }
   ```

2. Configure the property values for the MicroTx library. See Configure Library Properties for Node.js Apps. To enable logging for Node.js applications, you must set additional properties. See Enable Logs for MicroTx Node.js Library.

3. Configure the MicroTx 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 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 "tmmlib-
   ```

```
node/lra/lra";
import { getHeaderValue } from 'tmmlib-node/util/trmutils';
```

5. Create a router object to handle requests in your program.

   Use the following code to create a router object named `flightSvcRouter`.

   ```
   const flightSvcRouter = Router();
   ```

6. Enter the URL of the MicroTx Saga coordinator. To get this attribute value, append `/lra-coordinator` to the URL that you use to access MicroTx. For example, if `https://tmm-app:9000/api/v1` is the MicroTx URL, then `lraCoordinatorUrl` is `https://tmm-app:9000/api/v1/lra-coordinator`.

   ```
   const lraCoordinateUrl = process.env.ORACLE_TMM_TCS_URL
   ```

7. 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 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 adds a link header for all the outgoing requests from the REST API endpoints that you have specified.

8. 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.

9. Save your changes.

8

Develop Applications with TCC

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 to confirm all the reservations.

Guidelines to develop custom applications that use the TCC transaction protocol

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 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 client libraries places the URI in MicroTx 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.

- Workflow to Develop Applications with TCC
  Use the following workflow as a guide to develop your applications to use MicroTx to manage TCC transactions.
- Configure Library Properties
  Provide configuration information for the MicroTx 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 REST Apps with TCC
  Use the MicroTx library with your Spring REST applications.
- Develop JAX-RS Apps with TCC
  The MicroTx 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 library with your Node.js applications.
- Develop Python Apps with TCC
  MicroTx client libraries for Python applications provides the functionality to initiate a new TCC transaction or to participate in an existing TCC transaction.

8.1 Workflow to Develop Applications with TCC

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

Task	Description	More Information
Install MicroTx	Install MicroTx and ensure that you can access it.	Workflow to Install and Use MicroTx
Provide configuration information for the MicroTx 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 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

8.2 Configure Library Properties

Provide configuration information for the MicroTx 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.

- `oracle.tmm.TcsUrl` for JAX-RS apps or `spring.microtx.coordinator-url` for Spring REST apps: Enter the URL to access the MicroTx application. See Access MicroTx. 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.PropagateTraceHeaders` for JAX-RS apps or `spring.microtx.propagation-active` for Spring REST 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. For other microservices, set this property to `true`.

The following example provides sample values for the MicroTx client library properties for a JAX-RS application.

```
oracle.tmm.TcsUrl = http://tmm-app:9000/api/v1
oracle.tmm.PropagateTraceHeaders = true
```

The following example provides sample values for the MicroTx client library properties for a Spring REST application.

```
spring:
  microtx:
    coordinator-url: http://tmm-app:9000/api/v1
    propagation-active: true
```

You can use the HTTP protocol if your application and MicroTx 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 JAX-RS apps.

```
export ORACLE_TMM_TCS_URL = http://tmm-app:9000/api/v1
export ORACLE_TMM_PROPAGATE_TRACE_HEADERS = true
```

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
```

Note that the environment variables names are case-sensitive.

8.3 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 ensures that all participant services either confirm or cancel the reservation. When MicroTx 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 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.

8.4 Develop Spring REST Apps with TCC

Use the MicroTx library with your Spring REST applications.

For information about the MicroTx 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 REST App as Transaction Participant

8.4.1 Configure Spring REST App as Transaction Initiator

1. Specify property values for the MicroTx library. See Configure Library Properties.
2. Include the following MicroTx library file as a maven dependency in the JDBC application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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 the 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. Autowire the Spring Boot `RestTemplate`, provided by the MicroTx library. Use this object 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.

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

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.

```
@TCC
public class myTransactionParticipantApp {
    // Service code that is specific to the transaction participant
service.

    @Autowired
    @Qualifier("MicroTxTccRestTemplate")
    RestTemplate restTemplate;

    @Autowired
    MicroTxTccClient tccClientService;
    ...
}
```

7. Save the changes.

8.4.2 Configure Spring REST App as Transaction Participant

1. Specify property values for the MicroTx library. See Configure Library Properties.
2. Include the following MicroTx library file as a maven dependency in the JDBC application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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 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 services.

8.5 Develop JAX-RS Apps with TCC

The MicroTx 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 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 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 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

8.5.1 Configure JAX-RS App as Transaction Initiator

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

1.
2. Include the MicroTx Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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>24.2</version>
  </dependency>
  ```

3. 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;
   ```

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 the 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.

8.5.2 Configure JAX-RS App as Transaction Participant

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

1. Include the MicroTx Java library file as a maven dependency in the application's `pom.xml` file. The following sample code is for the 24.2 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>24.2</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>24.2</version>
     </dependency>
     ```

2. Import the following packages.

   ```
   import oracle.trm.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 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.

```
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 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.

8.6 Develop Node.js Apps with TCC

Use the MicroTx library with your Node.js applications.

For information about the MicroTx 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
`ConfirmTCC(req.headers);`	Confirms the current TCC transaction.
`CancelTCC(req.headers);`	Cancels the current TCC transaction.
`GetTCCId(req.headers)`	Get the current TCC transaction ID.

Topics

- Configure Node.js App as Transaction Initiator
- Configure Node.js App as Transaction Participant

8.6.1 Configure Node.js App as Transaction Initiator

1. Add the MicroTx library for Node.js as a dependency in the `package.json` file. The library file is located in the `installation_directory/otmm-RELEASE/otmm/nodejs` folder.

```
"dependencies": {
    "tmmlib-node": "file:oracle-microtx-1.0.0.tgz"
  }
```

2. Configure the property values for the MicroTx library. See Configure Library Properties. To enable logging for Node.js applications, you must set additional properties. See Enable Logs for MicroTx Node.js Library.
3. Configure the MicroTx 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 libraries and the express module files.

```
import {HttpMethod, TrmConfig} from "tmmlib-node/util/trmutils";
import {TCCConfig} from "tmmlib-node/tcc/tcc";
import {NextFunction, request, Request, Response, Router} from 'express';
```

5. Create a router object.

   Use the following code to create a router object named `svcRouter`.

```
const svcRouter = Router();
```

6. 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 the 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 adds a header for all the outgoing requests from the REST API endpoint that you have specified.

8.6.2 Configure Node.js App as Transaction Participant

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

1. Add the MicroTx library for Node.js as a dependency in the `package.json` file. The library file is located in the *`installation_directory`*`/otmm-`*`RELEASE`*`/otmm/nodejs` folder.

   ```
   "dependencies": {
       "tmmlib-node": "file:oracle-microtx-1.0.0.tgz"
     }
   ```

2. Configure the property values for the MicroTx library. See Configure Library Properties. To enable logging for Node.js applications, you must set additional properties. See Enable Logs for MicroTx Node.js Library.

3. Configure the MicroTx 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 libraries and the express module files.

   ```
   import {HttpMethod, TrmConfig} from "tmmlib-node/util/trmutils";
   import {TCCConfig} from "tmmlib-node/tcc/tcc";
   import {NextFunction, request, Request, Response, Router} from 'express';
   ```

5. Create a router object.

Use the following code to create a router object named `svcRouter2`.

```
const svcRouter2 = Router();
```

6. 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.

7. 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 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
   }));
   ```

8. 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 library enlists the participant service with the URIs you provide for the confirm and cancel endpoints.

8.7 Develop Python Apps with TCC

MicroTx 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 to manage a TCC transaction, update your Python application code to integrate the functionality provided by the MicroTx client libraries. For information about the MicroTx 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
`ConfirmTCC(`*`incoming_request_headers`*`);`	Confirms the current TCC transaction and returns the HTTP response.
`CancelTCC(`*`incoming_request_headers`*`);`	Cancels the current TCC transaction and returns the HTTP response.
`GetTCCId(`*`incoming_request_headers`*`)`	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

8.7.1 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 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 library file for Python which is available in the `installation_directory/otmm-<version>/lib/python` folder.

   ```
   pip3 install tmmpy-<version>.whl
   ```

2. Configure the property values for the MicroTx 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 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 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 the 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.

8.7.2 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 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 library file for Python which is available in the `installation_directory/otmm-<version>/lib/python` folder.

   ```
   pip3 install tmmpy-<version>.whl
   ```

2. Configure the property values for the MicroTx 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 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 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 the 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 library enlists the participant service with the URIs you provide for the confirm and cancel endpoints.

9

Develop Tuxedo Apps with XA

Enable Transaction Manager for Microservices (MicroTx) interoperability in your Tuxedo or SALT applications. You can only use the XA transaction protocol for your Tuxedo or SALT applications.

9.1 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.

9.1.1 Prepare the Environment

Ensure that network connectivity is there between the Tuxedo application and MicroTx.

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 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.

To verify bi-directional network connection between the Tuxedo application and MicroTx:

1. Use SSH to login to the MicroTx host and the Linux host on which you have installed the Tuxedo application.

2. Start a simple HTTP server on the MicroTx host.

   ```
   python -m SimpleHTTPServer 2345
   ```

   Where, `2345` is a port in the MicroTx 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 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.
   - `2345` is a port in the MicroTx 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 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.

9.1.2 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.

9.1.3 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, 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.
6. 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"
   "ubbshm"
   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.

7. Run the following commands to rebuild and start the sample Tuxedo application.

```
rm -f TLOG GWTLOG tuxconfig saltconfig bankdl1 bankdl2 bankdl3
. ./bankvar
tmloadcf -y ubbshm
wsloadcf -y bankapp.dep
./crbank
./crtlog
tmboot -y
./populate
```

8. 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, 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.

9.2 Run Tuxedo App in Kubernetes Cluster

Run the Tuxedo application in the same Kubernetes cluster in which you have deployed MicroTx.

- 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.

9.2.1 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.

9.2.2 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*/otmm-*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.

10

Manage the Transaction Coordinator

If you have installed Transaction Manager for Microservices (MicroTx) 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 server the 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.
- Scale up or down
  Scale up the Kubernetes cluster on which you have installed MicroTx to handle a large amount of requests. When the number of requests is low, scale down to use the resources efficiently.
- Update
- Uninstall

10.1 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 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 library injects the `oracle-tmm-txn-id` HTTP header in the outgoing request. All subsequent calls from MicroTx to the participant service also include this header. In this manner all requests are routed to a single 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. 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 a single 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 transaction coordinator

You must enable session affinity for the transaction coordinator in Saga and XA transactions, 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 a single replica of the transaction coordinator. You don't need to enable session affinity for TCC transactions.

The process to enable session affinity for the transaction coordinator and participant service is similar. To enable session affinity for the transaction coordinator, you will update the YAML files for the transaction coordinator.

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.

10.1.1 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. 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. To enable session affinity for the transaction coordinator, update the YAML files and Helm Chart that are specific to the transaction coordinator.

Before you begin, complete the following tasks:

1. Ensure that you have deployed the transaction participant service 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. Create a networking rule for the application in the namespace where you want to deploy it. The traffic policy must use a load balancer with consistent hash that uses the HTTP request header, `oracle-tmm-txn-id`.
2. In the Helm Chart of the participant application, specify the `oracle-tmm-txn-id` HTTP header in Istio's `DestinationRule` resource. Use a load balancer that is based on consistent hash to provide session affinity based on the `oracle-tmm-txn-id` HTTP header.

   ```
   apiVersion: networking.istio.io/v1alpha3
   kind: DestinationRule
   metadata:
       name: sticky-participant
       namespace: otmm
   spec:
       host: sticky-participant.otmm.svc.cluster.local
       trafficPolicy:
         loadBalancer:
           consistentHash:
             httpHeaderName: oracle-tmm-txn-id
   ```

   Where,

   - *`sticky-participant`* is the name of the participant application.
   - *`otmm`* is the namespace in which you want to deploy your participant application.
   - *`host`*: Specify the fully qualified name of your application inside the Kubernetes cluster. For example, `dept1.otmm.svc.cluster.local`.
3. In the `values.yaml` file of the participant service, add the following line of code:

   ```
   sessionAffinity: true
   ```
4. In the `networking.yaml` file of the participant service, add the following lines of code:

   ```
   spec:
       host: {{$val.host}}
       trafficPolicy:
         loadBalancer:
           consistentHash:
             httpHeaderName: oracle-tmm-txn-id
   ```

10.2 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 for the coordinator. See Enable Session Affinity.

10.3 About Transaction Recovery

The transaction coordinator server resumes the transactions that were in progress when server the 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 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 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.

10.4 General Syntax of Commands

The following is the general syntax of the Helm commands that you can run to manage MicroTx.

```
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.

Action	Description
upgrade	Updates one or more values that you have defined for MicroTx in its `values.yaml` file.
uninstall	Removes MicroTx from the Kubernetes cluster.

Command Parameter

Parameter	Description
`release_name`	Enter the name of the MicroTx 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. Example, `otmm`.
`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 application. Example, *`installation_directory`*`\otmm-`*`RELEASE`*`\otmm\helmcharts`.

10.5 Scale up or down

Scale up the Kubernetes cluster on which you have installed MicroTx 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 instance that you want to run at a time.

Perform the following steps to scale the Kubernetes cluster on which you have installed MicroTx:

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 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 `otmm` namespace with the details mentioned in the `scale.yaml` file.

```
helm upgrade otmm --namespace otmm --reuse-values --values scale.yaml otmm-
RELEASE\otmm\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.

10.6 Update

Use this command to update one or more property values that you have defined for MicroTx 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.
   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.

10.7 Uninstall

When you no longer want to use MicroTx, you can uninstall it from the Kubernetes cluster.

Prerequisites

Before you run this command, ensure that you do not need to run MicroTx 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 from the Kubernetes cluster. Communication between your applications microservices will continue, but any requests sent by the application microservices to MicroTx will fail.

Examples

Use the following command to delete the `tmm_app` application.

```
helm uninstall tmm_app
```

11 Manage Transactions Using the Web Console

Use the MicroTx 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 the transaction which MicroTx 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 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 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 transaction coordinator in the `values.yaml` file, but you can't edit it using the MicroTx 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

11.1 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.

Note:

Administrators can access the MicroTx 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 Specify the Admin Role in YAML file and Create a Secret with Identity Provider Client Credentials.

To access the MicroTx 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 console is `https://192.0.2.1/consoleui/`.

2. Enter the MicroTx console URL in any browser.

   The MicroTx 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.

11.2 View Details of a Transaction

The **Transactions** tab displays the transaction which MicroTx 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 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.

Transaction Detail	Description
Status	The current status of the transaction.
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 creates a unique branch ID when each participant service enlists to MicroTx. For example, when the participant service enlists, it provides `http://dept1:8081` as the URL to the MicroTx coordinator. When a service connects to multiple resource managers, MicroTx creates a unique branch ID for each resource manager.

11.3 View the Total Number of Transactions

You can view the total number of transaction requests that MicroTx has received since you had installed it and the number of transactions in different states.

1. Sign in to the MicroTx 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 was installed.
   - **Processed** displays the total number of transactions that MicroTx 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 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.

11.4 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 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 is currently processing.

11.5 View Coordinator Health

View the number of replicas of MicroTx that are in the running state and the percentage of CPU and memory that each replica currently consumes.

1. Sign in to the MicroTx 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.

11.6 View Console Health

1. Sign in to the MicroTx 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.

11.7 View Configuration Properties

You can view the property values that you have set for MicroTx transaction coordinator in the `values.yaml` file, but you can't edit it using the MicroTx 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 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 Configure the values.yaml File.

11.8 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.

11.8.1 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 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**.

11.8.2 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 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**.

11.8.3 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 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**.

11.9 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.

11.9.1 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 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**.

11.9.2 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 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**.

11.10 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.

11.10.1 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 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**.

11.10.2 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 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**.

12 Deploy Sample Applications

Using samples is the fastest way for you to get familiar with MicroTx. This is an optional task. After installing MicroTx, you may want to run sample applications and then use distributed tracing to understand how requests flow between MicroTx and the microservices.

Sample applications are microservices that demonstrate how you can develop your services for participating in different transaction protocols using MicroTx. The code of the sample applications includes the MicroTx libraries. You can use the sample applications as a reference while integrating the MicroTx 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

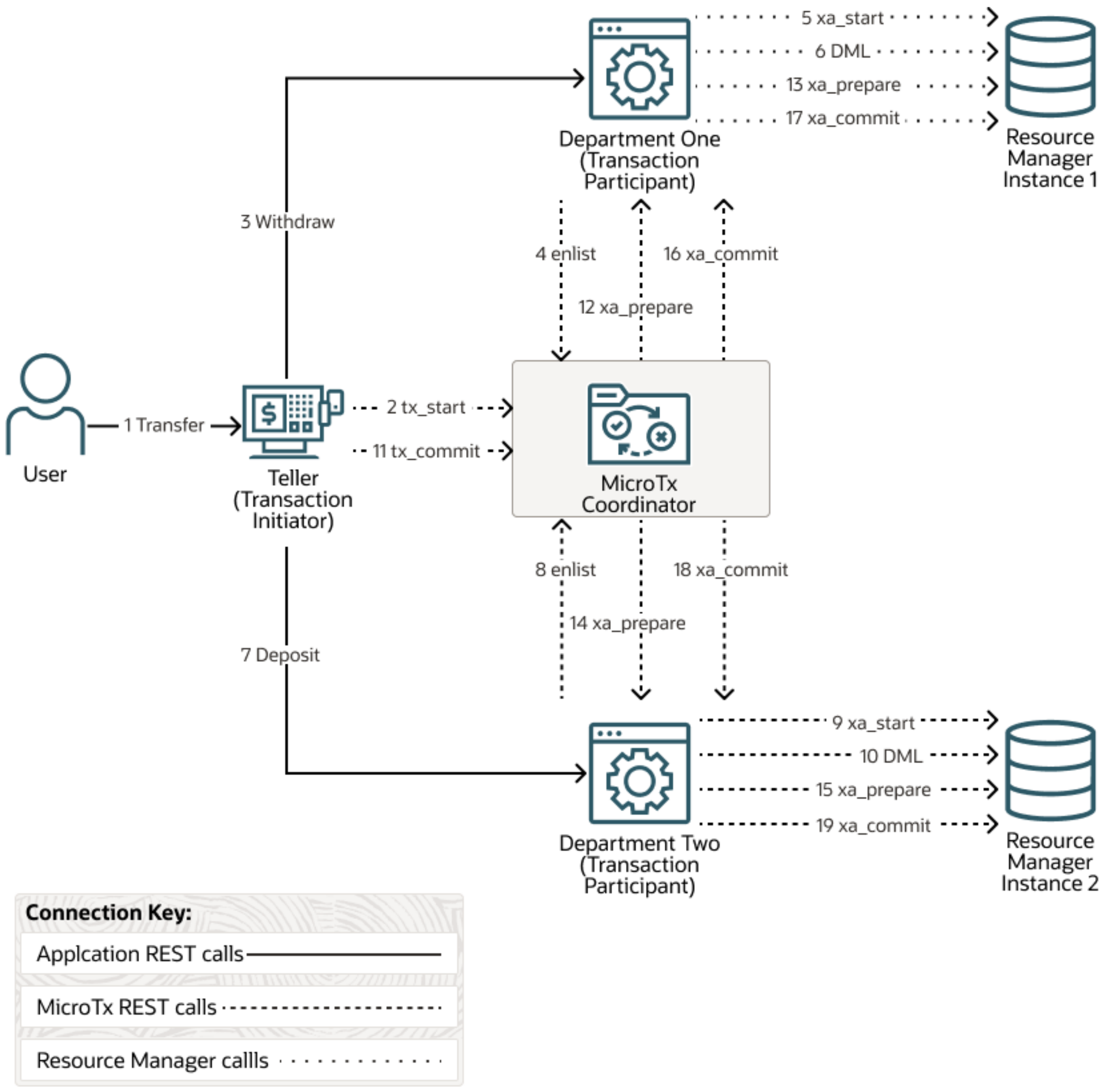
- About XA Sample Application
  The sample XA application implements a scenario where a Teller application initiates the transfer of an amout from one department to another by creating an XA transaction. The two departments in the organization are Department One (Dept 1) and Department Two (Dept 2).
- 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 interact with each other in a TCC transaction.

12.1 About XA Sample Application

The sample XA application implements a scenario where a Teller application initiates the transfer of an amout 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 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 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 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 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 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.

12.1.1 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.

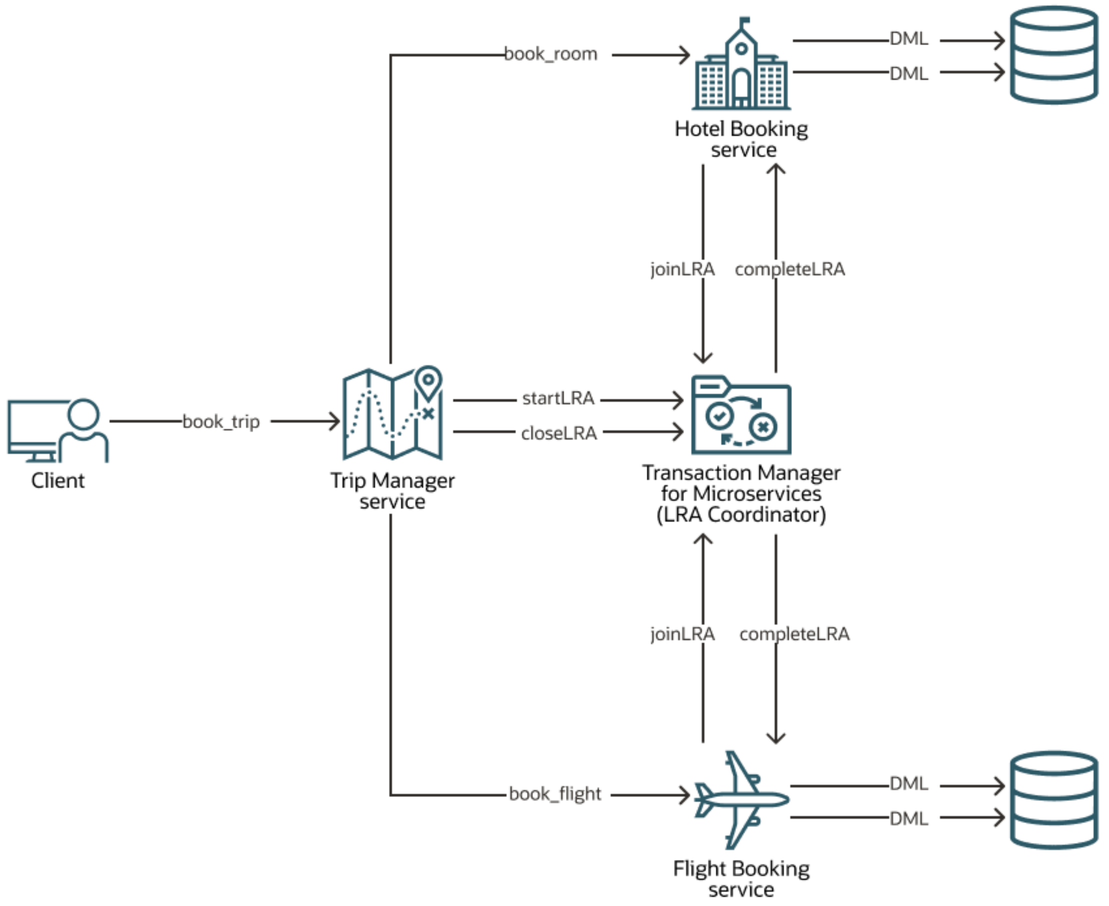
1. User places a request to transfer an amount from Department One to Department Two.
2. The Teller service initiates the transaction, when a user places a request to withdraw an amount from Department One. The transaction initiator service, Teller, makes a call to MicroTx to begin an XA transaction.
   MicroTx 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 with the same GTRID. MicroTx may have to interact with multiple participant services to successfully complete a transaction. MicroTx 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 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 with the same GTRID. MicroTx coordinator creates a branch ID that is unique to Department Two.

9. Department Two communicates with the resource manager using the integrated MicroTx 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 the 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 coordinator prepares the participant service, Department One, to commit the transaction.
13. MicroTx coordinator calls Department One. The participant microservice in turn uses the integrated MicroTx library to send a request to prepare the resource manager.
14. MicroTx coordinator prepares the participant service, Department Two, to commit the transaction.
15. Coordinator send a request to Department Two. The participant microservice in turn uses the integrated MicroTx 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 library. The coordinator returns a response to the Teller service which completes the transaction.

12.2 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 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 coordinates the transactions between these microservices.



The sample Saga application consists of the following polyglot microservices:

- MicroTx (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 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 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.

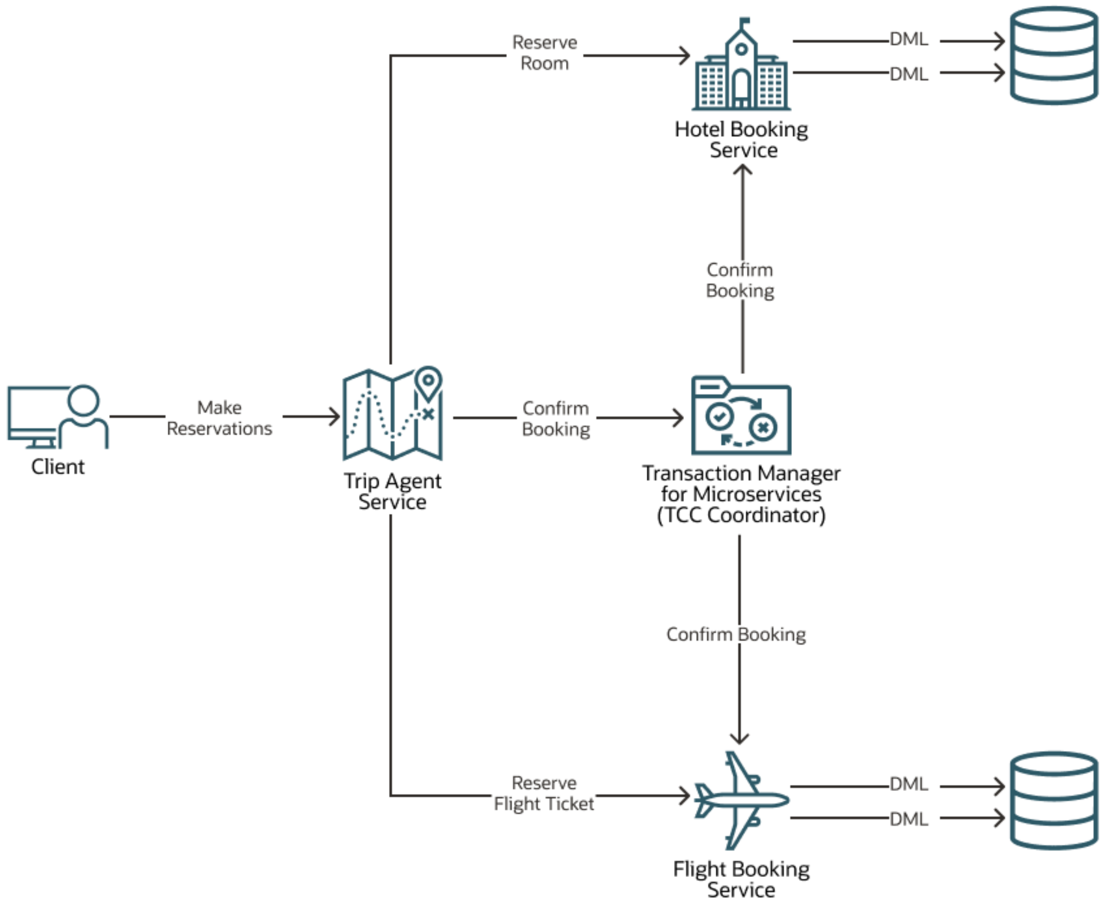
12.3 About the Sample TCC Application

Let's use the sample TCC application that's available in the installation bundle to understand how microservices and MicroTx interact with each other in a TCC transaction.

The sample application uses the TCC transaction protocol and MicroTx to coordinate the transactions. The MicroTx 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 to manage TCC transactions.



The sample TCC application consists of the following microservices:

- MicroTx (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.

13 Monitor and Trace Transactions

Monitor transactions to understand how the requests flow between MicroTx and the microservices and how MicroTx 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 and the microservices. Use tools, such as Kiali and Jaeger, to track and trace distributed transactions in MicroTx.
- 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 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 coordinator and the MicroTx client libraries for Java, Spring Boot, Node.js, and Python generate logs.

13.1 Trace

Use distributed tracing to understand how requests flow between MicroTx and the microservices. Use tools, such as Kiali and Jaeger, to track and trace distributed transactions in MicroTx.

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 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.

13.1.1 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-wqxhb                  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
REFERENCE                         TARGETS         MINPODS   MAXPODS
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
```

13.1.2 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`.

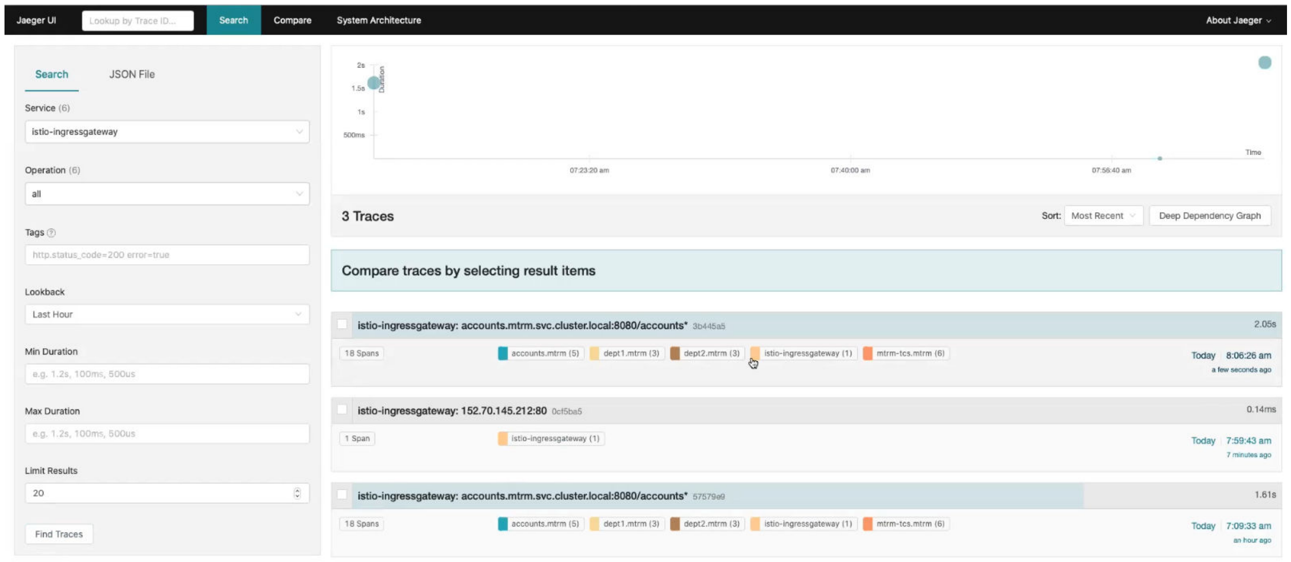
   ```
   istioctl dashboard jaeger
   ```

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.



Service (6): istio-ingressgateway
Operation (6): all
Tags: http.status_code=200 error=true
Lookback: Last Hour
Min Duration: e.g. 1.2s, 100ms, 500us
Max Duration: e.g. 1.2s, 100ms, 500us
Limit Results: 20

3 Traces — Sort: Most Recent

Compare traces by selecting result items

istio-ingressgateway: accounts.mtrm.svc.cluster.local:8080/accounts* 3b445a5 — 2.05s
18 Spans — accounts.mtrm (5), dept1.mtrm (3), dept2.mtrm (3), istio-ingressgateway (1), mtrm-tcs.mtrm (6) — Today 8:06:26 am, a few seconds ago

istio-ingressgateway: 152.70.145.212:80 0cf5ba5 — 0.14ms
1 Span — istio-ingressgateway (1) — Today 7:59:43 am, 7 minutes ago

istio-ingressgateway: accounts.mtrm.svc.cluster.local:8080/accounts* 57579e9 — 1.61s
18 Spans — accounts.mtrm (5), dept1.mtrm (3), dept2.mtrm (3), istio-ingressgateway (1), mtrm-tcs.mtrm (6) — Today 7:09:33 am, an hour ago

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.

13.1.3 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
   ```

13.1.4 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-specific headers must be propagated for running the MicroTx 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.

13.2 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 coordinator.

The MicroTx coordinator exposes metrics in a format that can be easily read by Prometheus. Prometheus sends a request to each MicroTx pod to collect metrics from each pod in the Kubernetes cluster. Prometheus monitors and gathers metrics about the MicroTx 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 dashboards to view metrics.

- Set Up Prometheus and Grafana
  Set up Prometheus and Grafana in the Kubernetes cluster where you have installed MicroTx to monitor MicroTx.
- View Metrics with Prometheus and Grafana
  View MicroTx coordinator metrics with Prometheus and Grafana.

13.2.1 Set Up Prometheus and Grafana

Set up Prometheus and Grafana in the Kubernetes cluster where you have installed MicroTx to monitor MicroTx.

Before you begin, ensure that you have installed Istio and MicroTx.

1. Set the `metrics` property to true in the `values.yaml` file, Helm Chart used to deploy the MicroTx coordinator. You must set this property to enable Prometheus to scrape the metrics logs of the MicroTx 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 host. Refer to the Prometheus documentation for more details.

   To ensure that Prometheus can discover the MicroTx host, check the Targets in Prometheus or search for any one of the MicroTx 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. Notw down the link to access the Grafana dashboard.

10. Create a Prometheus data source in Grafana, so that the MicroTx metrics scraped by Prometheus are displayed in the Grafana dashboard.

11. Import MicroTx dashboard by importing the JSON file which contains metrics and details about MicroTx. 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.

13.2.2 View Metrics with Prometheus and Grafana

View MicroTx 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 coordinator.

   Prometheus provides value for different metrics and a description for each metric.

3. Visualize the metrics scraped by Prometheus in the MicroTx dashboard in Grafana.

13.3 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 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 `prometheus.yaml`, which is located at .
3. Open `prometheus.yaml`, which is located at `*installation_directory*/otmm-*RELEASE*/otmm/visualization/alert/kubernetes`, and 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 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.

     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/.

4. 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
   ```

5. Reinstall Prometheus using the `prometheus.yaml` file that you have downloaded.

   ```
   helm install -f prometheus.yaml prometheus prometheus-community/prometheus
   ```

6. Download the Prometheus Alertmanager from https://prometheus.io/download/#alertmanager.
7. 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. Refer to the Prometheus Alertmanager documentation.

13.4 Logs

The MicroTx coordinator and the MicroTx client libraries for Java, Spring Boot, Node.js, and Python generate logs.

To view the logs emitted by MicroTx 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 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 Node.js Library
  By default, logging is enabled for the MicroTx Coordinator and Java applications that use the MicroTx client library. However, for Node.js applications that use the MicroTx client libraries, you must enable logging.

13.4.1 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 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 coordinator and applications at a single place. 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  mtxnodejslib
           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  mtxlraspringbootlib
           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+(?<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 Java library, add the MDC key `microtx-txnId` in `log4j.PatternLayout` to specify the syntax of the log messages.

   The following code sample demonstrates how to create the MDC context `microtx-txnId` and specify the position of the various fields in the log message. In this example configuration for Log4J, the name of the logger is `oracle.tmm`, the log level is set to `DEBUG`, and provide the `AppenderRef` as `MicroTxLib`. The `PatternLayout` specifies the syntax of the log messages. The `%X{microtx-txnId}` refers to the unique transaction ID. When MicroTx logs a message, `%X{microtx-txnId}` is replaced with the actual transaction ID.

```
<?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 : %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 : %X{microtx-txnId} : %msg%n%throwable"/>
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="oracle.tmm" level="DEBUG" additivity="false">
            <AppenderRef ref="MicroTxLib"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

If you want to use another tool to log the information, provide the syntax of the log message in a similar way.

13.4.2 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 Coordinator

The MicroTx 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 Java Client Libraries

As shown in the following snippet, every log message generated by the MicroTx 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.

```
2023-09-28 13:08:36.877  INFO  --- [helidon-2]
oracle.tmm.jta.filter.TrmTransactionResponseFilter :
          ccdd6bb2-1376-4d21-9e40-125dc4eaedb3 : TrmTransactionResponseFilter
Response Status : 200
```

Where,

- `2023-09-28 13:08:36.877` is the time stamp in YYYY-MM-DD HH:MM:SS.SSS format.
- `INFO` is the log level.
- `helidon-2` is the name of the Java application for which the log is generated.
- `oracle.tmm.jta.filter.TrmTransactionResponseFilter` is the name of the class for which the log is generated.
- `ccdd6bb2-1376-4d21-9e40-125dc4eaedb3` is the unique ID for a transaction.
- `TrmTransactionResponseFilter Response Status : 200` 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 Node.js Client Libraries

As shown in the following snippet, every log message generated by the MicroTx 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.

13.4.3 Enable Logs for MicroTx Node.js Library

By default, logging is enabled for the MicroTx Coordinator and Java applications that use the MicroTx client library. However, for Node.js applications that use the MicroTx client libraries, you must enable logging.

In the `tmm.properties` file, where you have provided values to configure the MicroTx 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 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.

14 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 coordinator
  Check the configuration details provided for coordinator and data store, and then redeploy the MicroTx coordinator.
- Unable to retrieve details about the MicroTx coordinator and console health
  Check if the Kubernetes Metrics Server is deployed.

14.1 Cannot retrieve transaction details from the MicroTx coordinator

Check the configuration details provided for coordinator and data store, and then redeploy the MicroTx coordinator.

Cause

An attempt was made to retrieve transaction details from the MicroTx coordinator, but the details could not be retrieved.

Action

Identify the issue, fix it, and then deploy the MicroTx 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 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 Privilege to Run Stored Procedures.
- Check that you have provided the correct configuration details for the MicroTx 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.

14.2 Unable to retrieve details about the MicroTx 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 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.

A
Deploy Your Application

Before you begin, ensure that you have completed the following tasks:

1. Installed Transaction Manager for Microservices (MicroTx).
2. Integrated MicroTx 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 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 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 `otmm` namespace.

   ```
   helm install my-java-app-tcc-tx --namespace otmm .\ --values .\values.yaml
   ```

   Where,

   - `my-java-app-tcc-tx` is the name of the application that you want to create.
   - `otmm` 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 `otmm` and their status.

   ```
   kubectl get all -n otmm
   ```

4. Verify that the application is installed.

   ```
   helm list --namespace otmm
   ```

B

Install on Docker Swarm

You can install Transaction Manager for Microservices (MicroTx) in Docker Swarm or in a Kubernetes cluster.

Follow the instructions in this section to install MicroTx in Docker Swarm and run sample applications. You can create a similar configuration to install MicroTx in other supported environments. If you want to install MicroTx on a Kubernetes cluster, skip this section and see Install on a Kubernetes Cluster.

Note:

The instructions provided in this section are specific to test or development environments. Do not use these instructions to set up and use Transaction Manager for Microservices in production environments.

- Set Up Docker Swarm
- Create a Registry
  Because a swarm consists of multiple Docker Engines, a registry is required to distribute images to all of them.
- Push Image to a Docker Registry
  The installation bundle that you have downloaded to your local system contains a Docker image of MicroTx. Push this image to the registry that you have created in Docker.
- Create Encryption Key and Key Pair
  Perform this task only if you want to enable the `authTokenPropagationEnabled` and `transactionTokenEnabled` properties in the `tcs-docker-swarm.yaml` file. This file is located in the `installation_directory/otmm-<`*`version`*`>/samples/docker` folder.
- Create a Docker Secret for Oracle Database Credentials
  MicroTx supports using Oracle Database as a persistent store to keep track of the transaction information. You must provide the Oracle Database credentials in the `YAML` file. MicroTx uses the credentials to establish a connection to the database after the service is installed.
- Update YAML files with etcd Details
  You must provide etcd credentials and etcd endpoints in the `YAML` files for the transaction coordinator. MicroTx uses this information to establish a connection to etcd after the service is installed.
- Configure the tcs-docker-swarm.yaml File
  The installation bundle contains `tcs-docker-swarm.yaml` file, the manifest file of the application, which contains the deployment configuration details for MicroTx.
- Configure Secure Connection for Your Apps
- Access MicroTx in Docker Swarm

B.1 Set Up Docker Swarm

1. Download and install Docker Desktop. See https://docs.docker.com/get-started/.
2. Run the following command in a shell prompt to ensure that the Docker engine is running in Swarm mode.

   ```
   docker system info
   ```

   In the response, scroll and search for the following message:

   ```
   Swarm: active
   ```

   If Swarm is not enabled, run the following command in a shell prompt to enable it.

   ```
   docker swarm init
   ```

3. Install a current version of Docker Compose. See https://docs.docker.com/compose/install/.

B.2 Create a Registry

Because a swarm consists of multiple Docker Engines, a registry is required to distribute images to all of them.

1. Run the following command to start the registry as a service on your swarm.

   ```
   docker service create --name registry --publish published=5000,target=5000
   registry:2
   ```

2. Run the following command to check the status of the registry.

   ```
   docker service ls
   ```

   In the response, look for `1/1` under `REPLICAS`, which indicates that the registry is running. If the response is `0/1`, it is probably still pulling the image. Check the status again after some time.

   ```
   ID             NAME               MODE         REPLICAS   IMAGE
   PORTS
   tjc0u55yavu4   registry           replicated   1/1        registry:2
   *:5000->5000/tcp
   ```

3. Verify that you can use cURL to access the registry.

   ```
   curl http://localhost:5000/v2/
   ```

B.3 Push Image to a Docker Registry

The installation bundle that you have downloaded to your local system contains a Docker image of MicroTx. Push this image to the registry that you have created in Docker.

Perform the following steps to push the Docker image of MicroTx to the registry in Docker:

1. Load the MicroTx image to the local Docker repository. The MicroTx image is located at `installation_directory/otmm-<version>/image/tmm-<version>.tgz`.

   ```
   cd installation_directory/otmm-<version>/otmm
   docker load < image/tmm-<version>.tgz
   ```

   The following message is displayed when the image is loaded.

   ```
   Loaded image: tmm:<version>
   ```

2. Create a tag for the image that you have loaded.
3. Use the following commands to specify a unique tag for the images 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.

   Sample Commands

   ```
   docker tag tmm:<version> 198.51.100.1:5000/tmm
   ```

   Where, `198.51.100.1:5000` is the Docker registry to which you want to push the image file, `tmm:<version>`. Provide the registry details based on your environment.

4. Push the Docker image with the new tag to the Docker registry.

   Syntax

   ```
   docker push remote_image[:tag]
   ```

   Sample Commands

   ```
   docker push 198.51.100.1:5000/tmm
   ```

B.4 Create Encryption Key and Key Pair

Perform this task only if you want to enable the `authTokenPropagationEnabled` and `transactionTokenEnabled` properties in the `tcs-docker-swarm.yaml` file. This file is located in the `installation_directory/otmm-<version>/samples/docker` folder.

If the `authTokenPropagationEnabled` and `transactionTokenEnabled` properties in the `tcs-docker-swarm.yaml` file need not be enabled, then you must comment a few lines in the two YAML files.

Comment the following lines in the `tcs-docker-swarm.yaml` file.

```
# secretKeys: '{"secretKeys":[{"secretKeyName":"TMMSECRETKEY",
"version":"1"}]}'
# EncryptionSecretKeyVersion: 1
```

```
...
# keyPairs: '{"keyPairs":[{"privateKeyName":"TMMPRIKEY",
"publicKeyName":"TMMPUBKEY", "version":"1",
"privateKeyPasswordName":"TMMPRIKEYPASSWD"}]}'
# transactionTokenKeyPairVersion: 1
```

Comment the following lines in the `tmm-stack-compose.yaml` file. This file is located in the `installation_directory/otmm-<version>/samples/docker` folder.

```
# secrets:
# TMMSECRETKEY:
# external: true
# TMMPRIKEY:
# external: true
# TMMPUBKEY:
# external: true
# TMMPRIKEYPASSWD:
# external: true

...
#entrypoint: ['/bin/sh', '-c', 'export TMMSECRETKEY=$$(cat /run/secrets/
TMMSECRETKEY); export TMMPRIKEY=$$(cat /run/secrets/TMMPRIKEY); export
TMMPUBKEY=$$(cat /run/secrets/TMMPUBKEY); export TMMPRIKEYPASSWD=$$(cat /run/
secrets/TMMPRIKEYPASSWD); /app/tcs' ]

# secrets:
    # - TMMSECRETKEY
    # - TMMPRIKEY
    # - TMMPUBKEY
    # - TMMPRIKEYPASSWD
```

Skip this section as you don't need to create encryption keys and transaction token as you have disabled these options.

You must generate an encryption key, and then add the key to a Docker secret if you have enabled the `authTokenPropagationEnabled` property under `authorization` in the `tcs-docker-swarm.yaml` file. The encryption key that you generate must have the following attributes.

- Symmetric algorithm: AES-256
- Cipher mode: AES in GCM mode
- Key length: 32 bytes
- Length of initialization vectors: 96 bits

You must generate a key pair for transaction token, when you set `transactionTokenEnabled` to `true` under `transactionToken` in the `tcs-docker-swarm.yaml` file. The transaction token that you generate must have the following attributes:

- Asymmetric algorithm: RSA 3072
- Key length: 3072 bits
- Hash algorithm: SHA256

You can reuse an existing RSA key, if you know the pass phrase. Otherwise, create a new RSA key.

Before you begin, ensure that you have installed OpenSSL.

For details about how the encryption token and transaction token are used, see About Authentication and Authorization.

To create an encryption key and a RSA key pair:

1. Run the following command to generate an encryption key with a key length of 32 bytes, and then create a secret while using the encrypted key.

   ```
   openssl rand -hex 16 | docker secret create TMMSECRETKEY
   ```

   Where, `TMMSECRETKEY` is the name of the secret that you want to create. If there is existing key with the same name that key is overwritten.

2. Create an RSA private key with key length as 3072 bits. Use the following command:

   ```
   openssl genrsa -aes256 -out private.pem 3072
   ```

3. Enter a pass phrase at the command prompt, and then press enter. Remember the pass phrase as you will have to provide it later.

   A new file called `private.pem` is created in the current working folder. This file contains the RSA private key value.

4. Create a RSA public key for the private key that you have generated.

   The following command creates a new file called `public.pem` in the current working folder. This file contains the RSA public key value.

   ```
   openssl rsa -in private.pem -outform PEM -pubout -out public.pem
   ```

5. Base-64 encode the private and public keys, and then add them to Docker secrets.

   ```
   base64 private.pem | docker secret create TMMPRIKEY -
   base64 public.pem | docker secret create TMMPUBKEY -
   ```

   Where, `TMMPRIKEY` and `TMMPUBKEY` are the names of the Docker secrets that you want to create.

6. Store the pass phrase for the RSA key as a Docker secret. In the following command, replace `pass_phrase` with the pass phrase for RSA key.

   ```
   printf "<pass_phrase>"| docker secret create TMMPRIKEYPASSWD -
   ```

7. View the names of the Docker secrets that you have created.

   ```
   docker secret ls
   ```

   Sample output

   ```
   ID                NAME                DRIVER    CREATED         UPDATED
   ricw56x6sehy...    TMMPRIKEY                     20 hours ago    20 hours ago
   c0hw2nhu0sh1...    TMMPRIKEYPASSWD               20 hours ago    20 hours ago
   mr91c79nwzne...    TMMPUBKEY                     20 hours ago    20 hours ago
   wp112txjki46...    TMMSECRETKEY                  20 hours ago    20 hours ago
   ```

   Note down the names of the keys as you'll need to provide it later.

8. Update the `tmm-stack-compose.yaml` file which is located in the `installation_directory/otmm-<version>/samples/docker` folder. Export the secrets that you have created as environment variables within the Swarm by providing details just below the `configs` section as shown in the following example.

```
version: "3.9"

configs:
   my_tcs_config:
   file: ./tcs-docker-swarm.yaml

secrets:
  TMMPRIKEY:
     external: true
  TMMPRIKEYPASSWD:
     external: true
  TMMPUBKEY:
     external: true
  TMMSECRETKEY:
     external: true
```

9. Add the following to the `services.otmm-tcs` section in the `tmm-stack-compose.yaml` file:
   - Names of the secrets that you have created.
   - Create an `entrypoint` to export the secrets that you have created as environment variables. To improve readability the following example uses same name for the secret and the environment variable. You can provide any other name for the environment variable. Note down the names of the environment variables as you will have to provide it in the next step.

```
services:
   otmm-tcs:
     image: "127.0.0.1:5000/tmm"
     ports:
        - "9000:9000"
     entrypoint: ['/bin/sh', '-c', 'export TMMPRIKEY=$$(cat /run/secrets/
TMMPRIKEY); export TMMPRIKEYPASSWD=$$(cat /run/secrets/TMMPRIKEYPASSWD);
export TMMPUBKEY=$$(cat /run/secrets/TMMPUBKEY); export TMMSECRETKEY=$$
(cat /run/secrets/TMMSECRETKEY); /app/tcs' ]
      deploy:
         replicas: 1
      configs:
        - source: my_tcs_config
          target: /tcs_config.yaml
      environment:
        - CONFIG_FILE=/tcs_config.yaml
      secrets:
        - TMMPRIKEY
        - TMMPRIKEYPASSWD
        - TMMPUBKEY
        - TMMSECRETKEY
```

10. Update the `tcs-docker-swarm.yaml` file with the names of the environment variables that you have created. This YAML file is located in the `installation_directory/otmm-<version>/samples/docker` folder.

Sample values for encryption and transactionToken properties

```
encryption:
     secretKeys: '{"secretKeys":[{"secretKeyName":"TMMSECRETKEY",
"version":"1"}]}'
     #TMMSECRETKEY is the environment variable for the Docker secret that
contains the encryption key
     EncryptionSecretKeyVersion: 1
transactionToken:
     transactionTokenEnabled: true
     keyPairs: '{"keyPairs":[{"privateKeyName":"TMMPRIKEY",
"publicKeyName":"TMMPUBKEY", "version":"1",
"privateKeyPasswordName":"TMMPRIKEYPASSWD"}]}'
     #TMMPRIKEY is the environment variable for the Docker secret that
contains the base64-encoded private key
     #TMMPUBKEY is the environment variable for the Docker secret that
contains the base64-encoded public key
     #TMMPRIKEYPASSWD is the environment variable for the Docker secret
that contains the private key password
     transactionTokenKeyPairVersion: 1
```

B.5 Create a Docker Secret for Oracle Database Credentials

MicroTx supports using Oracle Database as a persistent store to keep track of the transaction information. You must provide the Oracle Database credentials in the `YAML` file. MicroTx uses the credentials to establish a connection to the database after the service is installed.

Skip this step if you are not using Oracle Database to store the transaction details of MicroTx.

If you are using an Autonomous Database instance, ensure that you have downloaded the wallet and noted the connection string before you begin with the following steps. See Get Autonomous Database Client Credentials.

To create a Docker secret to provide the Oracle Database login details:

1. Enter the Oracle Database credentials in the following format in any text editor, such as Notepad. Replace the sample values with values that are specific to your environment.

```
{
  "password": "enter_your_Database_password",
  "username": "enter_the_username_to_access_the_Database"
}
```

2. Save the file with a `TXT` format. For example, `database_secret.txt`. Note down the path and name of this file as you'll need to provide it in the next step.
3. Create a Docker secret with the Oracle Database login details.

   Command syntax

```
docker secret create <name_of_the_secret> </path_to_text_file>/
<name_of_text_file
```

   Sample command

The following commands creates a Docker secret with the name `STORAGE_DB_CREDENTIAL`.

```
docker secret create STORAGE_DB_CREDENTIAL /database_secret.txt
```

4. Run the following command to verify that the secret has been created.

```
docker secret ls
```

Sample response

```
ID          NAME                    DRIVER     CREATED          UPDATED
ovn1x...    STORAGE_DB_CREDENTIAL              11 seconds ago   11 seconds
ago
```

To improve readability, the sample value in the response is truncated with `...`. When you run this command in your environment, you'll see the complete value.

Note down the name of the Docker secret that you have created. You will need to provide this name later.

5. Open the `tmm-stack-compose.yaml` file in any text editor. This file is located in the `installation_directory/otmm-<version>/samples/docker` folder.

6. Update the `otmm-tcs` service and `secrets` sections with the details of the Docker secret that you have created. The following code snippet shows sample values.

```
secrets:
  STORAGE_DB_CREDENTIAL:
    external: true
services:
  otmm-tcs:
    image: "127.0.0.1:5000/tmm"
    ports:
      - "9000:9000"
    deploy:
      replicas: 1
    configs:
      - source: my_tcs_config
        target: /tcs.yaml
    # Create an environment variable that points to the Docker secret that
you have created.
    entrypoint: ['/bin/sh', '-c', 'export STORAGE_DB_CREDENTIAL=$$
(cat /run/secrets/STORAGE_DB_CREDENTIAL); /app/tcs' ]
    environment:
      - CONFIG_FILE=/tcs.yaml
    secrets:
      - STORAGE_DB_CREDENTIAL
```

Where, `STORAGE_DB_CREDENTIAL` is the name of the Docker secret that you have created. Add an `entrypoint` to create an environment variable that points to the Docker secret that you have created. The name of the environment variable and the Docker secret are the same in the sample code snippet.

7. Enter the database connection string. Only if you are using an Autonomous Database instance, you must also specify the wallet details in the `volumes` parameter. For details about the format of the connection string for Autonomous Database instance, see Get Autonomous Database Client Credentials.

```
secrets:
  STORAGE_DB_CREDENTIAL:
    external: true
services:
  otmm-tcs:
    image: "127.0.0.1:5000/tmm"
    ports:
      - "9000:9000"
    deploy:
      replicas: 1
    configs:
      - source: my_tcs_config
        target: /tcs.yaml
    volumes:
      - /<PATH_TO_DOWNLOADED_WALLET>/<WALLET_FOLDER_NAME>:/app/Wallet
    entrypoint: ['/bin/sh', '-c', 'export STORAGE_DB_CREDENTIAL=$$
(cat /run/secrets/STORAGE_DB_CREDENTIAL); /app/tcs' ]
    environment:
      - CONFIG_FILE=/tcs.yaml
    secrets:
      - STORAGE_DB_CREDENTIAL
storage:
    type: db
    #Allowed types - etcd/db/memory
    db:
      connectionString: tcps://adb.us-ashburn-1.oraclecloud.com:1522/
bfeldfxbtjvtddi_brijeshadw1_medium.adb.oraclecloud.com?
retry_count=20&retry_delay=3&wallet_location=/app/Wallet
```

B.6 Update YAML files with etcd Details

You must provide etcd credentials and etcd endpoints in the `YAML` files for the transaction coordinator. MicroTx uses this information to establish a connection to etcd after the service is installed.

Skip this step if you are not using etcd to store the transaction logs of MicroTx.

Before you begin, generate RSA certificates for server and client. Create a JSON file with the contents of the generated certificates. See Generate RSA Certificates for etcd.

To create Docker secret with details to access etcd:

1. Update the `tcs-docker-swarm.yaml` file, provide the etcd endpoint, path to the credentials for etcd, and path to the RSA certificates for etcd. The following code snippet provides sample values used in Generate RSA Certificates for etcd. Replace these sample values with the actual values in your environment.

```
storage:
  type: etcd
  etcd:
    endpoints: https://etcd:2379
    credentialsFilePath: "/app/etcd/etcdecred.json"
    cacertFilePath: "/app/etcd/ca.pem"
    skipHostNameVerification: false
```

For reference information about each field, see Transaction Store Properties.

2. Update the `tcs-stack-compose.yaml` file with details about `etcd` under `services`.

   Sample values

   The following code snippet provides sample values used in Generate RSA Certificates for etcd and it considers that etcd and the transaction coordinator are in the same network in a Docker Swarm.

   Replace these sample values with the actual values in your environment.

```
services:
  etcd:
    image: "bitnami/etcd"
    ports:
      - "2379:2379"
      - "2380:2380"
    volumes:
      - <PATH_TO_CFSSL_DIRECTORY>/cfssl:/etcdssl
    environment:
      - ETCD_ROOT_PASSWORD=password
      - ETCD_CERT_FILE=/etcdssl/server.pem
      - ETCD_KEY_FILE=/etcdssl/server-key.pem
      - ETCD_LISTEN_CLIENT_URLS=https://0.0.0.0:2379
      - ETCD_ADVERTISE_CLIENT_URLS=https://127.0.0.1:2379
```

   Where,

   - `image` is the path to the etcd image file.
   - `ports` are the ports through which etcd communicates with the transaction coordinator.
   - `volumes` is the unique path to the etcd volume in Docker Swarm. Each service in Docker Swarm uses its own volume. MicroTx creates this volume during the installation process and copies the certificate files from your local directory to the volume. Specify the name in the following format: `<`*`absolute_path_to_certificate_directory_in_your_local_machine`*`>:/<`*`unique_name_of_etcd_volume`*`>`. For example, `<`*`PATH_TO_CFSSL_DIRECTORY`*`>/cfssl:/etcdssl`.
   - `ETCD_ROOT_PASSWORD` is an environment variable required by etcd. It is the password to access etcd.
   - `ETCD_CERT_FILE` is an environment variable required by etcd. It is the path to the server public key file in the etcd service volume in Docker Swarm. Specify the name in the following format: `<`*`unique_name_of_etcd_volume`*`>/<`*`name_of_server_certificate`*`>`. For example, `/etcdssl/server.pem`.
   - `ETCD_KEY_FILE` is an environment variable required by etcd. It is the path to the server private key file in the etcd service volume in Docker Swarm. Specify the name in the following format: `<`*`unique_name_of_etcd_volume`*`>/<`*`name_of_server_private_key_file`*`>`. For example, `/etcdssl/server-key.pem`.

- `ETCD_LISTEN_CLIENT_URLS` is an environment variable required by etcd. Specify the value in the following format: `<etcd_IP_address>/<etcd_port>`. For example, `https://0.0.0.0:2379` if etcd and the transaction coordinator are in the same network in Docker Swarm. In case, you have set up etcd is a separate network, specify the IP address of etcd. `2379` is the port used for communication with etcd. You have specified the ports that etcd uses under `ports`.
- `ETCD_ADVERTISE_CLIENT_URLS=` is an environment variable required by etcd. Specify the value in the following format: `<etcd_IP_address>/<etcd_port>`. For example, `https://127.0.0.1:2379`. In case, you have set up etcd is a separate network, specify the IP address of etcd in place of `127.0.0.1`. `2379` is the port used for communication with etcd. You have specified the ports that etcd uses under `ports`.

3. Add details about the absolute path to the directory that contains the certificates under `otmm-tcs` in the `tcs-stack-compose.yaml` file.

   The following sample code shows a snippet of the entries under `otmm-tcs`.

   ```
   otmm-tcs:
      volumes:
        - <PATH_TO_CFSSL_DIRECTORY>/cfssl:/app/etcd
   ```

   Where, `/app/etcd` is the unique path to the transaction coordinator volume in Docker Swarm. Each service in Docker Swarm uses its own volume. MicroTx creates this volume during the installation process, and then copies the certificate files from your local directory to the volume. Specify the name in the following format: `<absolute_path_to_certificate_directory_in_your_local_machine>:/<unique_name_of_transaction_coordinator_volume>`. For example, `<PATH_TO_CFSSL_DIRECTORY>/cfssl:/app/etcd`.
4. Save the changes.

B.7 Configure the tcs-docker-swarm.yaml File

The installation bundle contains `tcs-docker-swarm.yaml` file, the manifest file of the application, which contains the deployment configuration details for MicroTx.

Replace the sample values in the `tcs-docker-swarm.yaml` file to provide the environment details, image details, and configuration details. The details that you provide are used to deploy MicroTx in Docker Swarm.

To provide configuration details for MicroTx:

1. Open the `tcs-docker-swarm.yaml` file in any code editor. This file is located in the `installation_directory/otmm-<version>/samples/docker` folder. This file contains sample values.
2. Replace the sample values with values that are specific to your environment.

   The tables in this section describe the properties for the environment, storage, authorization, authentication, and other configuration details that are required to deploy MicroTx.
3. Save your changes.

- Transaction Coordinator Properties
  Provide information to configure MicroTx.

- Transaction Store Properties
  MicroTx uses a transaction store for persistence of transaction state.
- TLS Properties
  Run MicroTx using the HTTP or HTTPS protocol.
- Authorization Properties
  MicroTx supports authorization across participant services and coordinator by propagating the JWT token in every request. Use the `authTokenPropagationEnabled` field to control this function. Configure your identity providers to auto-refresh the expired access tokens at the coordinator.
- Authentication Properties
  Enter values for the `issuer` and `jwksUri` parameters of the JSON Web Token (JWT) which is used for authentication. To find information for these fields, use the Discover URL.
- Encryption Key Properties
  Under `encryption`, specify the encryption key that MicroTx uses to encrypt the access and refresh tokens. You must provide values for these properties if you have enabled the `authTokenPropagationEnabled` property under `tmmConfiguration.authorization`.
- Transaction Token Properties
  Under `transactionToken`, specify the key pair that you want to use for transaction token.

B.7.1 Transaction Coordinator Properties

Provide information to configure MicroTx.

Property	Description
`tmmAppName`	Enter the name of the MicroTx application that you want to create. When you install MicroTx, the MicroTx application is created with the name that you specify. Note down this name as you will need to provide it later. For example, `tmm-app`.
`listenAddr`	Enter the port over which you want to access MicroTx. Create the required networking rules to permit inbound and outbound traffic on this port. Note down this number as you will need to provide it later. For example, `0.0.0.0:9000`. Specify the listener address in the format, `<IP_address>:<port>`, as provided in the example.
`internalAddr`	Enter the internal URL to access MicroTx from within the Docker repository where you will install the service. See Access MicroTx in Docker Swarm.
`externalUrl`	Enter the external URL to access MicroTx from outside the Docker repository where you will install the service. See Access MicroTx in Docker Swarm.
`httpClientTimeoutInSecs`	Specify the maximum amount of time, in seconds, for which the HTTP callback API requests sent by the MicroTx coordinator to the participant services remains active. Enter an integer between 0 to 900. The default value is 180 seconds and the maximum value is 900 seconds. If you set this value to 0, then MicroTx does not enforce any limit. When the coordinator sends a HTTP callback API request to the participant services, the participant services must respond within the time period that you specify. If the participant service does not respond within the specified time period, the HTTP request sent by the coordinator times out.
`xaCoordinator.enabled`, `lraCoordinator.enabled`, or `tccCoordinator.enabled`	Set `enabled: true` for the transaction protocols that you want to use. MicroTx supports three distribution transaction protocols: XA, Saga, and TCC. If you want to use nest an XA transaction within a Saga transaction, set `enabled: true` for both `xaCoordinator` and `lraCoordinator`.

Property	Description
`xaCoordinator.txMaxTimeout`	Only for the XA transaction protocol. 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 is 600000 ms.
`narayanaLraCompatibilityMode.enabled`	Only for the Saga transaction protocol. Set this property to `true` when you want to use Saga participant applications that were implemented to work with the Narayana LRA Coordinator and now would participate in Saga transactions using MicroTx. Enable this mode to ensure that the MicroTx Saga APIs return the same response data that Narayana LRA Coordinator APIs return.
`logging.level`	Enter one of the following types to specify the log level for MicroTx: • `info`: Logs events that occur during the normal operation of the MicroTx. 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 to indicate that there is an issue that requires troubleshooting. • `debug`: Logs all the events. Use this setting when you want to debug an issue.
`logging.httpTraceEnabled`	Set this to `True` to log all the HTTP request and responses in MicroTx when you want to debug. If you set this to `True`, you must also set the `logging: level:` to `debug`.
`logging.devMode`	Set this to `True` only in test environments to get more details for debugging purposes. You must set this to `False` in production environments.
`maxRetryCount`	The maximum number of times that the transaction coordinator retries sending the same request again in case of any failures. For example, 10.
`minRetryInterval`	The minimum interval, in milliseconds, after which the transaction coordinator retries sending the same request again in case of any failures. The default value is 1000 ms.
`maxRetryInterval`	The maximum retry interval, in milliseconds, before which the transaction coordinator retries sending the same request again in case of any failures. For example, 10000.
`skipVerifyInsecureTLS`	Oracle recommends that you set this value to `false` and set up a valid certificate signed by trusted authorities for secure access. When you set this value to `false`, the transaction coordinator accesses the participant applications over the HTTPS protocol with a valid certificate signed by trusted authorities. The default value is `false`. If you set this value to `true`, the transaction coordinator can access the participant application's callback URL, without a valid SSL certificate, in an insecure manner. **Caution:** Do not set this value to `true` in production environments.

B.7.2 Transaction Store Properties

MicroTx uses a transaction store for persistence of transaction state.

You can use an etcd cluster, Oracle Database, or internal memory for storing transaction information. When you want to use multiple replicas of the transaction coordinator or in production environments, you must set up an etcd cluster or Oracle database as the transaction store. Use internal memory only for development environments as all the transaction details are lost every time you restart MicroTx. If you use internal memory, you can't create multiple replicas of the transaction coordinator.

Type of Transaction Store

Under `tmmConfiguration.storage`, specify the type of transaction store that MicroTx uses for persistence of transaction state. After specifying the type of transaction store, you can provide additional details to connect to the external transaction store.

Property	Description
`type`	Enter one of the following values to specify the persistent data that you want MicroTx to use to track the transaction information. • `etcd` to use etcd as the transaction store. You must provide details to connect to the etcd transaction store in the `storage: etcd:` field. • `db` to use Oracle Database as the transaction store. You must provide details to connect to the Oracle transaction store in the `storage: db:` field. • `memory` to skip entering details to connect to either etcd or Oracle Database and use the internal memory instead. When you use internal memory, all the transaction details are lost every time you restart MicroTx. If you want to use multiple replicas of the transaction coordinator while using the internal memory as transaction store, you must enable session affinity.

Oracle Database as Transaction Store

Under `tmmConfiguration.storage.db`, specify the details to connect to an Oracle Database. Skip this section and do not provide these values if you are connecting to an etcd database or using internal memory.

For details about creating the required Docker secret, see Create a Docker Secret for Oracle Database Credentials.

Property	Description
`connectionString`	Enter the connection string to the transaction 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: `<publicIP>:<portNumber>/<database unique name>.<host domain name>` For example, `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 Database, then enter a connection string similar to the following example: `jdbc:oracle:thin:@tcps://adb.us-phoenix-1.oraclecloud.com:7777/unique_connection_string_low.adb.oraclecloud.com&wallet_location=/app/Wallet`.

Property	Description
`netServiceName`	Enter the name of the Docker secret that contains the credentials to connect to the Oracle Database. Example, `db-secret`.

etcd Database as Transaction Store

Under `tmmConfiguration.storage.etcd`, specify the details to connect to an etcd database. Skip this section and do not provide these values if you are connecting to an Oracle database or using internal memory.

Property	Description
`endpoints`	Enter the URL to access etcd as a Docker Swarm service. For example, `https://etcd:2379` if etcd and the transaction coordinator are in the same network in Docker Swarm. Where, `2379` is the port used for communication with etcd. In case, you have set up etcd is a separate network, specify the IP address of etcd.
`skipHostNameVerification`	Set this to `false` to verify the IP address of the etcd database server. If you set this to `true`, then the server host name or IP address is not verified. You can set this field to `true` only for test or development environments. **Caution:** You must set this field to `false` in production environments.
`cacertFilePath`	Enter the path to the `ca.pem` file, certificate that you have created earlier. For example, `/app/etcd/ca.pem`.
`credentialsFilePath`	Enter the location of the JSON file, that contains client credentials, client key, and the password that you have used to protect the client certificate. For example, `/app/etcd/etcdecred.json`.

B.7.3 TLS Properties

Run MicroTx using the HTTP or HTTPS protocol.

For secure access to MicroTx over HTTPS, create a self-signed certificate and note down location of the certificate and private key. For information about creating an SSL certificate, see Guidelines for Generating Self-Signed Certificate and Private Key using OpenSSL in *Security Guide*.

If you enable TLS in the `tcs-docker-swarm.yaml` file, then you must import the SSL certificate into the trust store of the sample applications so that sample applications can securely access MicroTx.

Under `tmmConfiguration.serveTLS`, specify the details of the SSL certificate that you want to use for authorization.

Property	Description
`enabled`	Set this to `true` to enable TLS to ensure secure communication between participant services and MicroTx. You must provide details for the certificate and key file under `certFile` and `keyFile` properties. When you enable TLS, you can access the transaction coordinator over HTTPS. If you set this field to `false`, you don't have to provide values for the `certFile` and `keyFile` properties. When you disable TLS, you can access the transaction coordinator over HTTP. You must provide the `internalAddr` and `externalUrl` using HTTP protocol. For example, `http://localhost:9000`. **Caution:** You must set this field to `true` in production environments.
`certFile`	Path to the TLS certificate, in PEM format, on your local machine.
`keyFile`	Path to the private key file, in PEM format, which is associated with the certificate on your local machine.

The following code snippet provides sample values for the `serveTLS` field in the `tcs-docker-swarm.yaml` file.

```
tmmConfiguration:
  serveTLS:
    enabled: true
    certFile: /users/john.doe/self-signed/tcs/certificate.pem
    keyFile: /users/john.doe/self-signed/tcs/key.pem
```

B.7.4 Authorization Properties

MicroTx supports authorization across participant services and coordinator by propagating the JWT token in every request. Use the `authTokenPropagationEnabled` field to control this function. Configure your identity providers to auto-refresh the expired access tokens at the coordinator.

Property	Description
`enabled`	Set this to `true` to enable MicroTx check the subject in the incoming JWT token. MicroTx then tags the subject or user against the transaction ID, and further changes to the transaction is allowed only by the tagged subject or user. If you set this field to `false`, you don't have to provide values for the other properties under `tmmConfiguration.authorization`. **Caution:** You must set this field to `true` in production environments.

Property	Description
`authTokenPropagationEnabled`	Set this to `true` to enable token propagation to ensure secure communication between participant services and MicroTx. When you enable token propagation, you must provide the details for the encryption keys under the `encryption` property in the `tcs-docker-swarm.yaml` file.
`IdentityProviderName`	Specify the identity provider that you are using. Permitted values are: `IDCS` for Oracle IDCS and Oracle IAM, `KEYCLOAK` for Keycloak, `AZURE_AD` for Azure Active Directory, and `MICROSOFT_AD` for Microsoft Active Directory.
`IdentityProviderUrl`	Specify the URL of the identity provider. This information is required to create a new access token by using the refresh token. If you do not provide this information, expired access tokens are not auto-refreshed.
`IdentityProviderClientId`	Specify the client ID of the identity provider. This information is required to create a new access token by using the refresh token. If you do not provide this information, expired access tokens are not auto-refreshed.

B.7.5 Authentication Properties

Enter values for the `issuer` and `jwksUri` parameters of the JSON Web Token (JWT) which is used for authentication. To find information for these fields, use the Discover URL.

When you enable authentication, the transaction coordinator enforces JWT-based authentication and validates the authentication token against the public key. You must pass the access token in the `authorization` header.

Property	Description
`enabled`	Set to `false` to bypass JWT authentication. This permits requests that do not have JWT tokens. Enter `true` if you want all requests to have a JWT token. MicroTx validates the token provided in the request and denies access if the token is invalid. If you set `enabled` as `true`, then you must provide values for the `issuer` and `jwksUri` parameters of the JWT. **Caution:** You must set this property to `true` in production environments.
`jwt.issuer`	Identifies the JWT token issuer.
`jwt.jwksUri`	The URL of the identity provider's publicly hosted `jwksUri`, which is used to validate signature of the JWT. The JSON Web Key Set (JWKS) contains the cryptographic keys which are used to verify the incoming JWT tokens.

The following code snippet provides sample values for `authentication` field in the `tcs-docker-swarm.yaml` file. The sample values in this example are based on the values used in the sample commands in Run the Discovery URL.

```
authentication:
  enabled: true
  jwt:
    issuer: "https://identity.oraclecloud.com"
    jwksUri: "https://idcs-a83e....identity.oraclecloud.com:443/admin/v1/
SigningCert/jwk"
```

B.7.6 Encryption Key Properties

Under `encryption`, specify the encryption key that MicroTx uses to encrypt the access and refresh tokens. You must provide values for these properties if you have enabled the `authTokenPropagationEnabled` property under `tmmConfiguration.authorization`.

Property	Description
`EncryptionSecretKeyVersion`	Specify the version of the key that you want to use for encrypting the transaction tokens.
`secretKeys.secretKeyName`	Specify the name of the environment variable which points to the Docker secret that contains the encryption key. To support the encryption keys rotation, you can specify multiple encryption keys and their versions.
`secretKeys.version`	Enter the version of the Docker secret that you want to use.

If you create a new Docker secret, do not delete the entry for the previous secret immediately. You may delete the old secret and the corresponding entry in the `tcs-docker-swarm.yaml` file after a few days because existing transactions may be using the older versions of the key. After a few days, you can update the `tcs-docker-swarm.yaml` file, and then update MicroTx.

The following code snippet provides sample values for the `encryption` field in the `tcs-docker-swarm.yaml` file. The sample values in this example are based on the values used in the sample commands in Create Encryption Key and Key Pair.

```
encryption:
  secretKeys: '{"secretKeys":[{"secretKeyName":"TMMSECRETKEY",
"version":"1"}]}'
  #TMMSECRETKEY is the environment variable that points to the Docker secret
that contains the encryption key.
  EncryptionSecretKeyVersion: 1
```

B.7.7 Transaction Token Properties

Under `transactionToken`, specify the key pair that you want to use for transaction token.

If you set `transactionTokenEnabled` to `true` in `tcs-docker-swarm.yaml`, you must provide values for the properties listed in the following table.

Property	Description
`transactionTokenEnabled`	Set this to `true` when you want MicroTx to include a signed transaction token, `tmm-tx-token`, in the request header. You don't have to create the `tmm-tx-token` transaction token or pass it in the request header. The MicroTx library creates this token based on the private-public key pair that you provide. For more information about creating the key pair, see Create Encryption Key and Key Pair.
`transactionTokenKeyPairVersion`	Enter the version of the key pair that you want to use for signing and verification of the transaction token. When there are multiple key pairs, you must specify the version of the key pair that you want to use.

Property	Description
`keyPairs.keyPairs.privateKeyName`	Enter the name of the Docker secret which has the base64-encoded value of the private key.
`keyPairs.keyPairs.publicKeyName`	Enter the name of the Docker secret which has the base64-encoded value of the public key.
`keyPairs.keyPairs.version`	Enter the version of the private-public key pair that you want to use.
`keyPairs.keyPairs.privateKeyPasswordName`	Enter the name of the Docker secret which has the value of the pass phrase that you had provided while generating the private key.

The following code snippet provides sample values for the `transactionToken` field.

```
transactionToken:
  transactionTokenEnabled: false
  keyPairs: '{"keyPairs":[{"privateKeyName":"TMMPRIKEY",
"publicKeyName":"TMMPUBKEY", "version":"1",
"privateKeyPasswordName":"TMMPRIKEYPASSWD"}]}'
     #TMMPRIKEY is the environment variable for the Docker secret that
contains the base64-encoded private key
     #TMMPUBKEY is the environment variable for the Docker secret that
contains the base64-encoded public key
     #TMMPRIKEYPASSWD is the environment variable for the Docker secret that
contains the private key password
  transactionTokenKeyPairVersion: 1
```

B.8 Configure Secure Connection for Your Apps

1. Provide configuration information for the MicroTx library properties for all participant and initiator 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` for JAX-RS apps or `spring.microtx.coordinator-url` for Spring REST apps: Enter the URL to access the MicroTx application. See Access MicroTx. 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 which MicroTx calls back. Provide this value in the following format:

     ```
     http://HostNameofApp:PortofApp/
     ```

     Where,

     – *`HostNameofApp`*: The host name of your initiator or participant service. For example, `host.docker.internal`.

     – *`PortofApp`*: The port number over which you can access your participant service. For example, `8080`.

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

```
oracle.tmm.TcsUrl = https://localhost:9000/api/v1
oracle.tmm.CallbackUrl = http://host.docker.internal:8080
```

2. For your Java microservices to access the transaction coordinator over TLS, you must import the TLS certificate into the JRE Keystore using keytool.

   ```
   export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-11.0.11.jdk/
   Contents/Homesudo keytool -import -trustcacerts -alias tcs-localhost -file
   localhost.pem -keystore $JAVA_HOME/lib/security/cacerts
   ```

3. For your Node.js microservices to access the transaction coordinator over TLS, set the `NODE_EXTRA_CA_CERTS` environment variable to the path a root certificate, in PEM format.

   ```
   export NODE_EXTRA_CA_CERTS=./rootCA.crt
   ```

   For information about creating an SSL certificate, see Guidelines for Generating Self-Signed Certificate and Private Key using OpenSSL in Security Guide.

Place the `YAML` file in the current directory along with the certificate and key files. If you have set `tmmConfiguration.serveTLS.enabled` to `true` in the `YAML` file to enable TLS, you must copy the certificate and key files into the current directory.

B.9 Access MicroTx in Docker Swarm

Use the internal URL or external URL to access MicroTx. You will use different URLs depending on whether you want to access MicroTx from within the Docker registry where you have deployed the service or from outside the Docker registry.

When you enable TLS, use the HTTPS protocol to access the service. When you disable TLS, use the HTTP protocol to access the service.

Internal URL to access MicroTx

Use the internal URL to access MicroTx from within the Docker registry where you have deployed the service.

To access MicroTx internally, create the URL in the following format:

```
http://internalHostname:listenAddr/api/v1
```

Where,

- *`internalHostname`*: Name that you have entered for the `tmmAppName` property in the `tcs-docker-swarm.yaml` file. For example, `tmm-app`.
- *`listenAddr`*: Port number that you have entered for the `listenAddr` property in the `tcs-docker-swarm.yaml` file. For example, `9000`. Ensure that you have set up the required networking rules to permit HTTPS or HTTP traffic over this port.

Based on the example values provided above, the example MicroTx URL is `http://tmm-app:9000/api/v1` or `http://localhost:9000/api/v1`.

All communication within a Docker registry uses the HTTP protocol.

External URL to access MicroTx

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

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

```
https://externalHostname:listenAddr/api/v1
```

Where,

- *`externalHostname`*: The IP address of the Docker registry that you have created. For example, `198.51.100.1`.
- *`listenAddr`*: Port number that you have entered for the `listenAddr` property in the `tcs-docker-swarm.yaml` of the MicroTx. For example, `5000`. Ensure that you have set up the required networking rules to permit inbound and outbound HTTPS or HTTP traffic over this port.

Based on the example values provided above, the example MicroTx URL is `https://198.51.100.1:5000/api/v1`.

Store the IP address of the Docker registry in an environment variable named `REGISTRY_IPADDR` as shown in the following command.

```
export REGISTRY_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.

C

Run MicroTx in a Docker Container

You can run MicroTx in an independent Docker container in Windows, macOS, and Linux operating systems.

Topics

- Run MicroTx in a Docker Container on macOS or Linux
  Use the following commands to run MicroTx in an independent Docker container on macOS (Intel x86) and Linux operating systems.
- Run MicroTx in a Docker Container on Windows
  Use the following commands to run MicroTx in an independent Docker container on Windows operating system. These commands have been tested on PowerShell.

C.1 Run MicroTx in a Docker Container on macOS or Linux

Use the following commands to run MicroTx in an independent Docker container on macOS (Intel x86) and Linux operating systems.

Before you begin, ensure that you have loaded the MicroTx Docker image and updated the `tcs.yaml` file. The `tcs.yaml` file is located at `installation_directory/otmm-<version>/otmm/image` in your local machine. This file contains the deployment configuration details for MicroTx. The properties in the `tcs.yaml` and `tcs-docker-swarm.yaml` files are similar. For information about the configuration details, see Configure the tcs-docker-swarm.yaml File.

Run MicroTx using the configuration details provided in the `tcs.yaml` file. Ensure that the `tcs.yaml` file is present in the current working directory.

1. Use the following command to run MicroTx in a Docker container if you have not set up a data store and want to use internal memory.

   Sample Command

   ```
   docker container run --name otmm -v "$(pwd)":/app/config \
   -w /app -p 9000:9000/tcp --env CONFIG_FILE=/app/config/tcs.yaml \
   --network=host -d tmm:<version>
   ```

   Where,

   - `--name otmm` is the name of the container that you want to create.
   - `-v "$(pwd)":/app/config` mounts the current directory into container at the `/app/config` path.
   - `-w /app` specifies the working directory as `/app`. In this sample command, `/app` remains the default working directory as the volume is also mounted on this directory based on the value provided for the `-v` flag.
   - `--env CONFIG_FILE=config/tcs.yaml` specifies the location of the `tcs.yaml` file, which contains coordinator configuration.
   - `--network=host` is a network configuration. In macOS, the callback URL is `host.docker.internal2`. In Linux operating systems, the callback URL is an IPv4

address. To retrieve the IPv4 address, run the `ipconfig` command and then note down the value of `IPv4 Address`.

- `tmm:<version>` is the MicroTx Docker image that you have loaded to the local Docker repository.

2. Use the following commands to run MicroTx in a Docker container when you want to use Oracle Database as a data store to persist the transaction state and to store the transaction logs.

   a. Create a copy of the `tcs.yaml` file and name it as `tcs-db.yaml`.

   b. Open the `tcs-db.yaml` file in any code editor. Under `tmmConfiguration.storage`, specify the `type` as `db`, and then provide details to connect to an Oracle Database under `tmmConfiguration.storage.db`. See Transaction Store Properties.

   c. Use the following command to run MicroTx in a Docker container while providing credentials to access Oracle Database. Replace *`<DBpassword>`* and *`<DBuser>`* with the password and user name to access Oracle Database in your environment.

      Sample Command

      ```
      docker container run --name otmm -v "$(pwd)":/app/config \
      -w /app -p 9000:9000/tcp --env CONFIG_FILE=/app/config/tcs-db.yaml \
      --network=host \
      -e STORAGE_DB_CREDENTIAL='{"password":"<DBpassword>",
      "username":"<DBuser>"}' \
      -d tmm:23.4.2
      ```

      Where,

      - `--name otmm` is the name of the container that you want to create.
      - `-v "$(pwd)":/app/config` mounts the current directory into container at the `/app/config` path.
      - `-w /app` specifies the working directory as `/app`. In this sample command, `/app` remains the default working directory as the volume is also mounted on this directory based on the value provided for the `-v` flag.
      - `--env CONFIG_FILE=config/tcs-db.yaml` specifies the location of the `tcs.yaml` file, which contains configuration details for the coordinator and Oracle Database credentials.
      - `--network=host` is a network configuration. In macOS, the callback URL is `host.docker.internal2`. In Linux operating systems, the callback URL is an IPv4 address.
      - `tmm:<version>` is the MicroTx Docker image that you have loaded to the local Docker repository.

3. After MicroTx is installed, run the MicroTx health check API to verify that the MicroTx coordinator is up and running.

   ```
   curl --location \
             'http://localhost:9000/health' \
             --header 'Accept: application/json'
   ```

What's next? Install and run sample applications in your environment. See Deploy Sample Applications.

C.2 Run MicroTx in a Docker Container on Windows

Use the following commands to run MicroTx in an independent Docker container on Windows operating system. These commands have been tested on PowerShell.

Before you begin, ensure that you have loaded the MicroTx Docker image and updated the `tcs.yaml` file. The `tcs.yaml` file is located at `installation_directory/otmm-<version>/otmm/image` in your local machine. This file contains the deployment configuration details for MicroTx. The properties in the `tcs.yaml` and `tcs-docker-swarm.yaml` files are similar. For information about the configuration details, see Configure the tcs-docker-swarm.yaml File.

Run MicroTx using the configuration details provided in the `tcs.yaml` file. Ensure that the `tcs.yaml` file is present in the current working directory.

1. Use the following command to run MicroTx in a Docker container if you have not set up a data store and want to use internal memory.

   Sample Command

   ```
   docker container run --name otmm -v "$(pwd)":/app/config \
   -w /app -p 9000:9000/tcp --env CONFIG_FILE=/app/config/tcs.yaml \
   --network=host -d tmm:<version>
   ```

   Where,

   - `--name otmm` is the name of the container that you want to create.
   - `-v "$(pwd)":/app/config` mounts the current directory into container at the `/app/config` path.
   - `-w /app` specifies the working directory as `/app`. In this sample command, `/app` remains the default working directory as the volume is also mounted on this directory based on the value provided for the `-v` flag.
   - `--env CONFIG_FILE=config/tcs.yaml` specifies the location of the `tcs.yaml` file, which contains coordinator configuration.
   - `--network=host` or `--add-host host.docker.internal:host-gateway`. Based on your requirements, select a network configuration.
     – If you specify `--network=host`, the callback URL is `host.docker.internal`.
     – If you specify `--add-host host.docker.internal:host-gateway`, the callback URL is an IPv4 address. To retrieve the IPv4 address, run the `ipconfig` command and then note down the value of `IPv4 Address`.
   - `tmm:<version>` is the MicroTx Docker image that you have loaded to the local Docker repository.

2. Use the following commands to run MicroTx in a Docker container when you want to use Oracle Database as a data store to persist the transaction state and to store the transaction logs.

   a. Create a copy of the `tcs.yaml` file and name it as `tcs-db.yaml`.

   b. Open the `tcs-db.yaml` file in any code editor. Under `tmmConfiguration.storage`, specify the `type` as `db`, and then provide details to connect to an Oracle Database under `tmmConfiguration.storage.db`. See Transaction Store Properties.

c. Store the credentials to access Oracle Database in an environment variable, `STORAGE_DB_CREDENTIAL`, as shown in the following example. Provide values based on your environment. You'll use this environment variable in the next step.

```
$env:STORAGE_DB_CREDENTIAL='{\"password\":\"<dbUserpassword>\",
\"username\":\"<DBuserName>\"}'
```

d. Use the following command to run MicroTx in a Docker container while providing credentials to access Oracle Database.

Sample Command

```
docker container run --name otmm -v "$(pwd)":/app/config \
-w /app -p 9000:9000/tcp --env CONFIG_FILE=/app/config/tcs-db.yaml \
--network=host \
-env STORAGE_DB_CREDENTIAL="$($env:STORAGE_DB_CREDENTIAL)" \
-d tmm:23.4.2
```

Where,

- `--name otmm` is the name of the container that you want to create.
- `-v "$(pwd)":/app/config` mounts the current directory into container at the `/app/config` path.
- `-w /app` specifies the working directory as `/app`. In this sample command, `/app` remains the default working directory as the volume is also mounted on this directory based on the value provided for the `-v` flag.
- `--env CONFIG_FILE=config/tcs-db.yaml` specifies the location of the `tcs-db.yaml` file, which contains coordinator configuration and credentials to connect to an Oracle Database.
- `--network=host` or `--add-host host.docker.internal:host-gateway`. Based on your requirements, select a network configuration.
  – If you specify `--network=host`, the callback URL is `host.docker.internal`.
  – If you specify `--add-host host.docker.internal:host-gateway`, the callback URL is an IPv4 address. To retrieve the IPv4 address, run the `ipconfig` command and then note down the value of `IPv4 Address`.
- `tmm:<version>` is the MicroTx Docker image that you have loaded to the local Docker repository.

3. After MicroTx is installed, run the MicroTx health check API to verify that the MicroTx coordinator is up and running.

```
curl --location \
          'http://localhost:9000/health' \
          --header 'Accept: application/json'
```

What's next? Install and run sample applications in your environment. See Deploy Sample Applications.