Oracle®

Transaction Manager for Microservices Developer Guide

Release 22.3

F48194-05

May 2023

Oracle Transaction Manager for Microservices Developer Guide, Release 22.3

F48194-05

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

Changes in MicroTx

1 About MicroTx

2 Plan

3 Prepare

9 Develop Applications with LRA

10 Develop Applications with TCC

11 Develop Tuxedo Apps with XA

12 Trace

A Manage Transaction Coordinator Using Helm

B Deploy Your Application

Changes in MicroTx

The following are the changes in the Transaction Manager for Microservices (MicroTx) patch 22.3.2 and the previous release.

- New Features in 22.3.2
  The following are the new features in the MicroTx release 22.3.2.
- Changes in the Previous Release
  The following are the changes in MicroTx release 22.3.1.

New Features in 22.3.2

The following are the new features in the MicroTx release 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 for Java and Node.js applications, TCC library for Python apps is now available. 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.

Changes in the Previous Release

The following are the changes in MicroTx release 22.3.1.

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 Databases.

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.

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, you can optimize the transaction. See Common Resource Manager for Multiple Apps.

Recover Transactions

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

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 deployed in Kubernetes clusters. It performs the following actions:

- Manages transactions and provides consistency across polyglot microservices.
- Supports several distributed transaction protocols, such as XA, 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 along with microservices applications running 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.

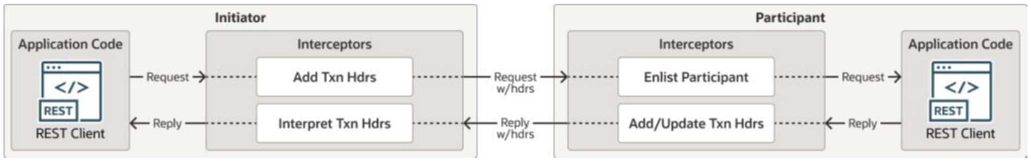
1.1 How MicroTx Works

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

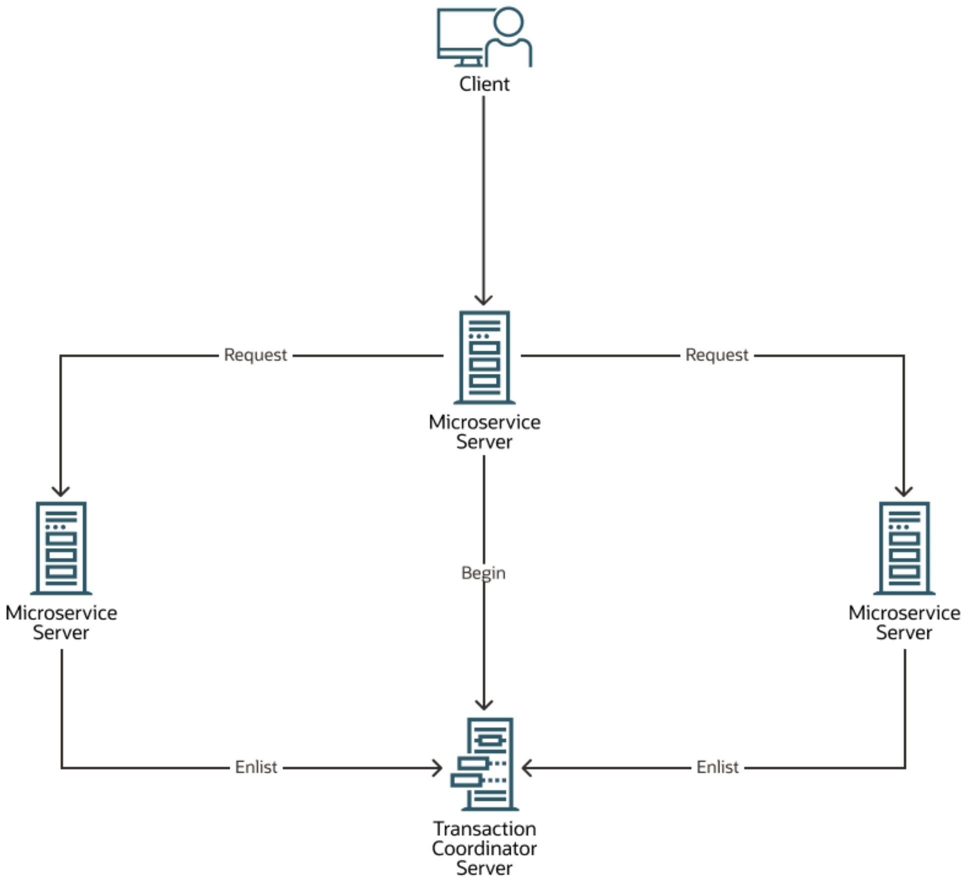
The MicroTx client libraries provide interceptors to intercept both incoming and outgoing REST calls, as well as their requests and responses. These interceptors use headers to propagate the transaction context which enable the participant microservices to automatically enlist in a transaction. The interceptors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

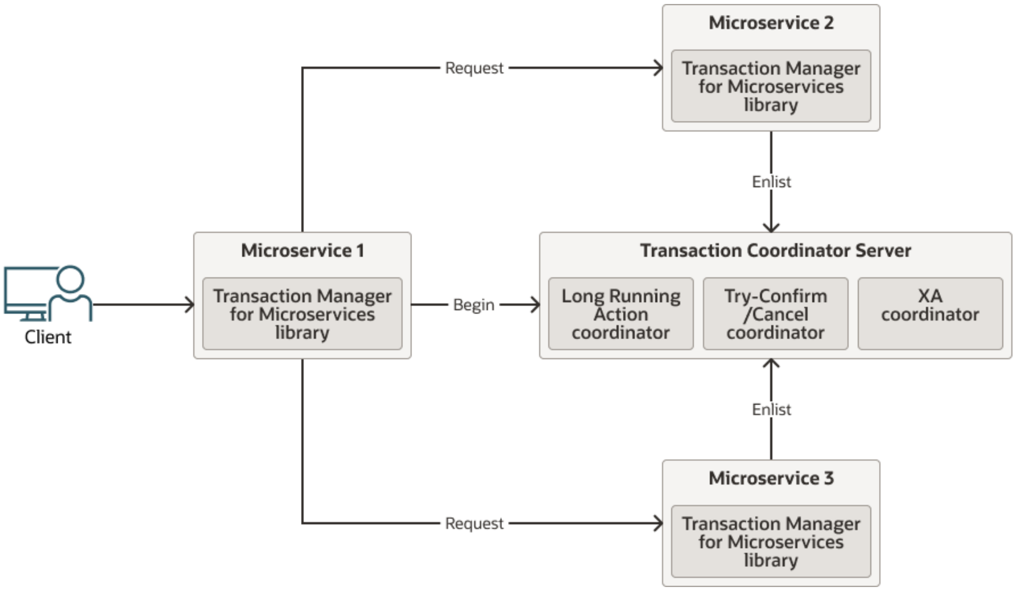


1. Application developers use functions present in the MicroTx library with their application code.
2. When a microservice or client initiates a transaction, it calls functions in the MicroTx library to start a distributed transaction.
3. MicroTx library includes headers that enable the participant services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

- Long Running Action (LRA) 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

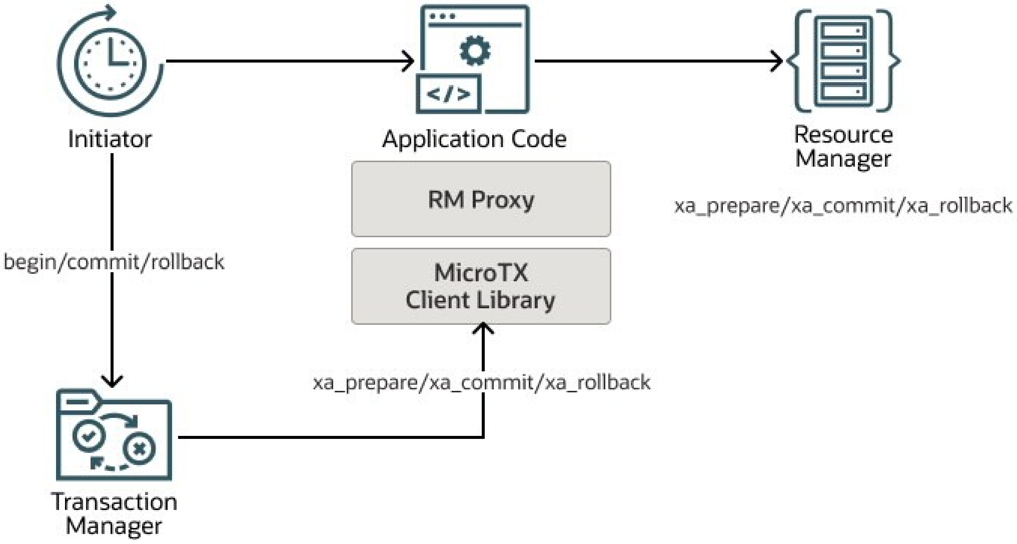
Use XA when strong consistency is required, similar to consistency provided by the local database transactions, where all the ACID properties of a transaction are present. For example, financial applications. Use the LRA protocol for transactions that may take a long time to complete. You can use the LRA 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 LRA and TCC support long running transactions. LRA is far more general, but requires application specific actions for both completing a successful LRA and compensating a failed LRA. 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.
- LRA 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.

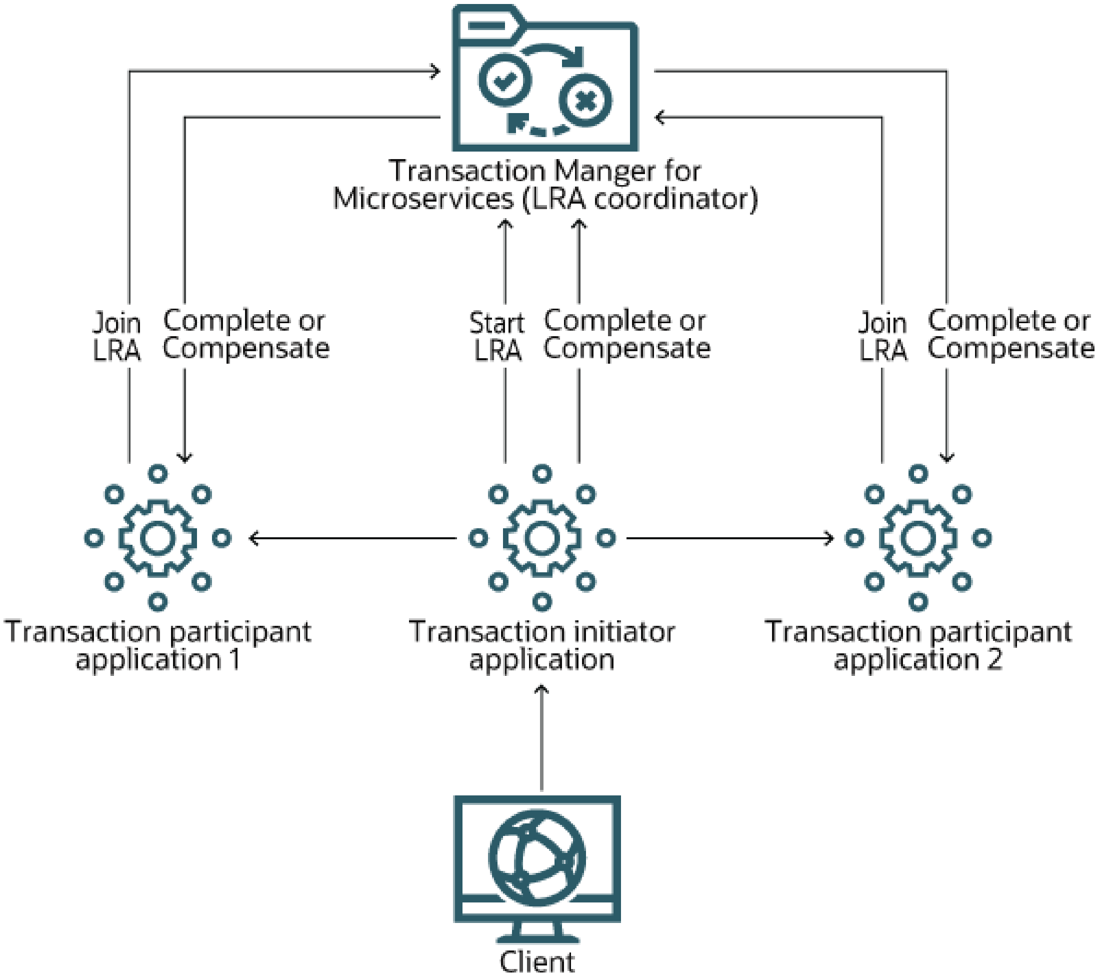


1. Initiator starts the distributed transaction
2. Called microservices enlist in the transaction
3. Initiator asks transaction manager to commit or rollback the transaction
4. If the initiator decided to commit, the transaction manager asks each microservice to prepare
    a. If all participants successfully 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 LRA Transaction Protocol

The following image describes how the microservices communicate with each other and with MicroTx when you use the Eclipse MicroProfile Long Running Actions (LRA) transaction protocol.


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 LRA coordinator and passes its callback URIs to begin and enlist in the LRA transaction.
2. The transaction initiator service calls one or more participant services by passing the ID of the LRA in headers.
3. The other participant services call MicroTx and enlist or join the LRA transaction. When participants join the LRA, they provide callback URIs including ones for completing and compensating their part of the LRA.
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 LRA 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.

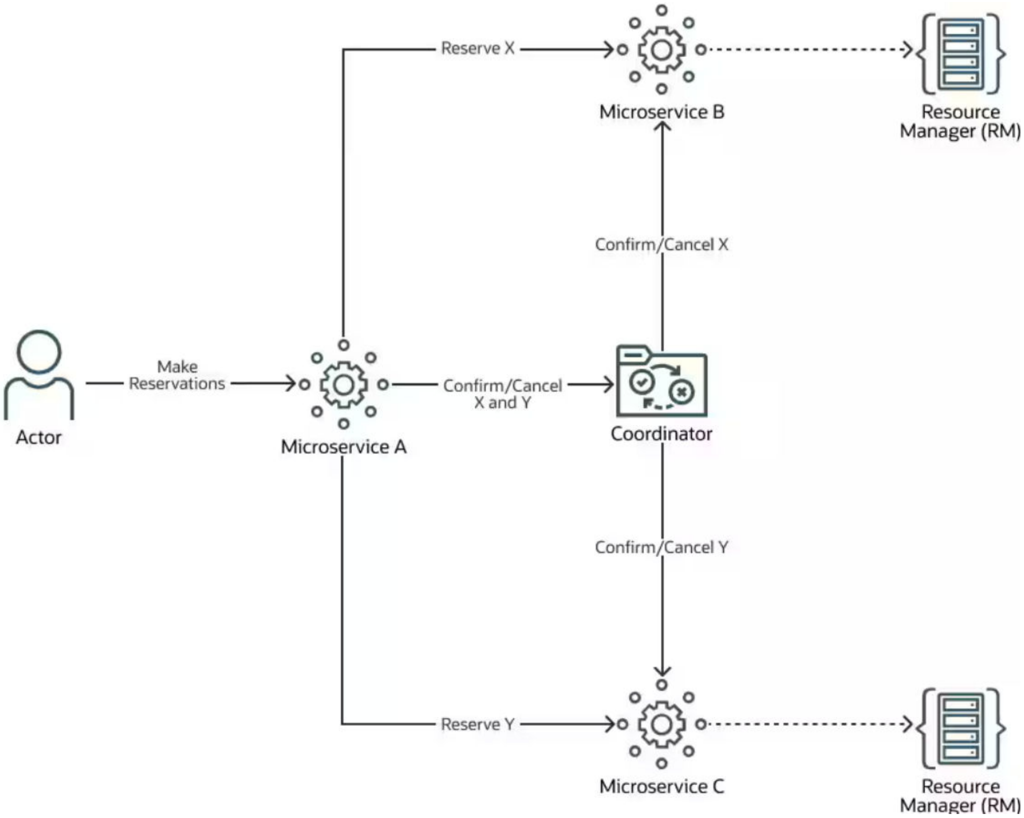
1.3.3 Try-Confirm/Cancel Transaction Protocol

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 LRA • 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).

2.1 Supported Container Platforms

You can deploy MicroTx on Docker or on Kubernetes cluster.

MicroTx is tested with Kubernetes 1.21.x. You can use any Kubernetes distribution compatible with Kubernetes 1.21.x.

MicroTx is tested with Docker 20.10.x. You can use any operating system that supports Docker 20.10.x or a compatible version.

2.2 Supported Languages

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

- TypeScript or JavaScript for Node.js
- Java (applications built with frameworks, such as Helidon, Spring Boot, and WebLogic Server)
- Python 3.3 or later

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

Java applications must use REST APIs implemented with Jersey. MicroTx libraries provides filters that are compatible with JAX-RS as implemented in Jersey.

2.3 Supported Databases

MicroTx release 22.3.1 supports the following databases:

- 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
- etcd

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

The transaction initiator service and transaction participant services may also use a database to store application data. If you select the XA transaction protocol for your application, see Supported Resource Managers for information about the resource managers that MicroTx supports. In XA transactions, MicroTx client libraries need to access the resource manager's client libraries.

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

2.4 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.

2.5 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.

2.6 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.

2.7 Select a Transaction Protocol

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 LRA may be a better fit. LRA 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	LRA	TCC
Transaction consistency level	Strongest	Eventual	Strong
Dirty reads	No	Yes	No

Parameters	XA	LRA	TCC
App development complexity	Low	High	Medium
Auto rollback on timeouts or errors	Yes	Yes	Yes
Transaction performance	Good	Better	Best
Locks held during the transaction	Yes	No	No

XA participants hold locks for the duration of the transaction. LRA 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 LRA 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. LRA 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 LRA 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 LRA places some significant burdens on the application. When an LRA 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 LRA and XA transaction protocols, you can nest an XA transaction within an LRA transaction. Let's consider an application which books movie tickets. The microservices that reserve the seats, use the LRA 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 LRA 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 Transaction Recovery

From MicroTx release 22.3.1, 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
LRA	`Closing`	reissues the close command to close the transaction
LRA	`Canceling`	reissues the cancel command to cancel the transaction
TCC	`Confirming`	reissues the confirm command to confirm the transaction

Transaction protocol	Transaction status	As part of transaction recovery, the transaction coordinator...
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.

2.9 About Session Affinity

MicroTx release 22.3.1 supports 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 LRA 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.

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.

3.1 Download the Installation Bundle

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

1. Download the MicroTx installation bundle (.zip file) from https://www.oracle.com/database/transaction-manager-for-microservices/.
2. Unzip the MicroTx installation bundle.

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

3. 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.

- `samples`: This folder contains the source code for sample applications for different transaction protocols: XA, LRA, and TCC. The source code of the sample applications also includes the MicroTx libraries.

3.2 Create a Data Store

Create a data store to store the transaction tables for MicroTx.

You can use either etcd or Oracle Database as the data store. Before installing MicroTx, you must install and configure the 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. MicroTx requires certain details about the database.

- Get Autonomous Database Client Credentials
  MicroTx supports using Oracle Database as a persistent store to keep track of the transaction information.
- 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.

3.2.1 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.

3.2.2 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`.

3.3 About Authentication and Authorization

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

- About Authentication and Authorization
  Use authorization and refresh tokens to ensure secure communication between the transaction initiator service and MicroTx. Store the access and refresh tokens to support asynchronous calls. Use token propagation to ensure secure communication between participant services and MicroTx.
- Use Oracle Identity Providers
  You can use Oracle Identity Cloud Service (IDCS) or Oracle IAM as an 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 About Authentication and Authorization

Use authorization and refresh tokens to ensure secure communication between the transaction initiator service and MicroTx. Store the access and refresh tokens to support asynchronous calls. Use token propagation to ensure secure communication between participant services and MicroTx.

- About Authorization and Refresh Tokens
  Use authorization and refresh tokens to ensure secure communication between the transaction initiator service and MicroTx. Use an identity provider to create an authorization 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 authorization and refresh tokens in the request header.
- About the Oracle_Tmm_Tx_Token Transaction Token
  Enable the creation and propagation of the transaction token to ensure secure communication between the participant services and MicroTx. 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 authorization and refresh tokens, and then uses it in asynchronous calls.

3.3.1.1 About Authorization and Refresh Tokens

Use authorization and refresh tokens to ensure secure communication between the transaction initiator service and MicroTx. Use an identity provider to create an authorization 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 authorization and refresh tokens in the request header.

Authorization 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 authentication 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 authorization token has the required system privileges to perform the operation. This ensures that only validated users can access the MicroTx APIs.

When you enable authorization in the YAML file and if you do not provide the authorization token when you send the request, the transaction is rejected as there is no authorization 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 LRA transaction protocol to book a hotel and flight. It can take a few minutes for the user to complete the bookings. However, the authentication 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.

3.3.1.2 About the Oracle_Tmm_Tx_Token Transaction Token

Enable the creation and propagation of the transaction token to ensure secure communication between the participant services and MicroTx. 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 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.

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.

3.3.1.3 About Encrypting and Storing Tokens

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

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 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.

3.3.2 Use 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.

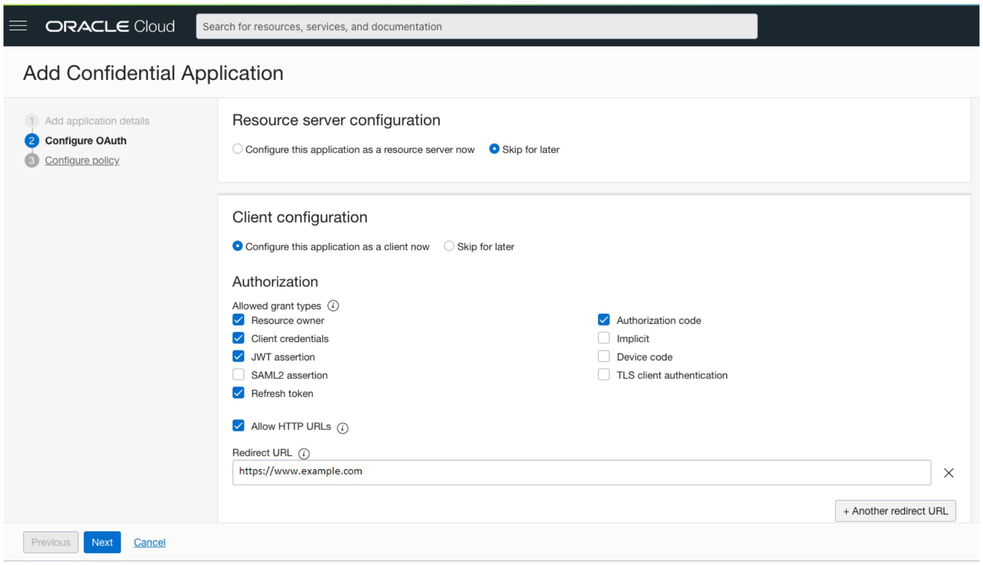
Based on whether your tenancy uses Oracle IDCS or Oracle IAM, you can use the relevant information to create a confidential application and activate it.

- 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.

3.3.2.1 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.

3.3.2.2 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.

For more information, see Token Validation in *REST API for Oracle Identity Cloud Service*.

To run the Discovery URL and note down the required 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-
   a83e4de370ea4db1c703a0b742ce74.identity.oraclecloud.com/.well-known/
   openid-configuration
   ```

   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-
   a83e4de370ea4db8c703a0b742ce74.identity.oraclecloud.com:443/
   admin/v1/SigningCert/jwk"
   ```
3. Note down the value for `audience`.

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 authentication 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. Launch a terminal and enter the following command.

   ```
   echo -n "clientid:clientsecret" | base64 -w 0
   ```

   Where, replace `clientid:clientsecret` with the values in your environment. `-w 0` is added for Linux to the command to remove line breaks.

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

   Based on your environment, you can use any base64 client to encode the `clientid:clientsecret`.

2. Copy the value that is returned. You'll have to provide this value every time you want to create an authentication token.

3. Get an authentication token using the base64-encoded value, as shown in the following cURL command example. Run one of the following commands based on whether you want to generate only the access token or the refresh token as well.

   - The following command creates the access token.

     Command syntax

     ```
     curl -i
     -H "Authorization:Basic {base64 encoded value of
     clientid:clientsecret}"
     -H "Content-Type: application/x-www-form-urlencoded;charset=UTF-8"
     --request POST https://domain-url/oauth2/v1/token
     -d
     "grant_type=password&username=username&password&scope=urn:opc:idm:__my
     scopes__"
     ```

     Example

     ```
     curl -i
     -H "Authorization:Basic ZWY1N2E1OWUyZjY..."
     -H "Content-Type: application/x-www-form-urlencoded;charset=UTF-8"
     --request POST https://idcs-
     ```

```
a83e4de370ea4db1b8c703a0b742ce74.identity.oraclecloud.com/
oauth2/v1/token
-d
"grant_type=password&username=acme@example.com&password&scope=urn
:opc:idm:__myscopes__"
```

- The following command creates the access token and the refresh token.

  Command syntax

  ```
  curl -i
  -H "Authorization:Basic {base64 encoded value of
  clientid:clientsecret}"
  -H "Content-Type: application/x-www-form-
  urlencoded;charset=UTF-8"
  --request POST https://domain-url/oauth2/v1/token
  -d
  "grant_type=password&scope=urn:opc:idm:__myscopes__+offline_acces
  s&username=username&password=password"
  ```

  Example

  ```
  curl -i
  -H "Authorization:Basic ZWY1N2E1OWUyZjY..."
  -H "Content-Type: application/x-www-form-
  urlencoded;charset=UTF-8"
  --request POST https://idcs-
  a83e4de370ea4db1b8c703a0b742ce74.identity.oraclecloud.com/
  oauth2/v1/token
  -d
  "grant_type=password&scope=urn:opc:idm:__myscopes__+offline_acces
  s&username=acme@example.com&password=password"
  ```

4. Copy the `access_token` value from the response as shown in the following example.

   Example output

   ```
   {
     "access_token":"eyJ4Lm...",
     "expires_in": 300,
     "refresh_expires_in": 1800,
     "refresh_token": "ey5Gkr...",
     "token_type": "Bearer",
     "not-before-policy": 0,
     "session_state": "c966d...",
     "scope": "profile email"
   }
   ```

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

   Make sure to copy only the actual token, which is the `access_token` and `refresh_token` values between the quotation marks.

5. Store the authentication token and refresh tokens in environment variables, as shown in the following example for a Linux host.

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

6. 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.

4

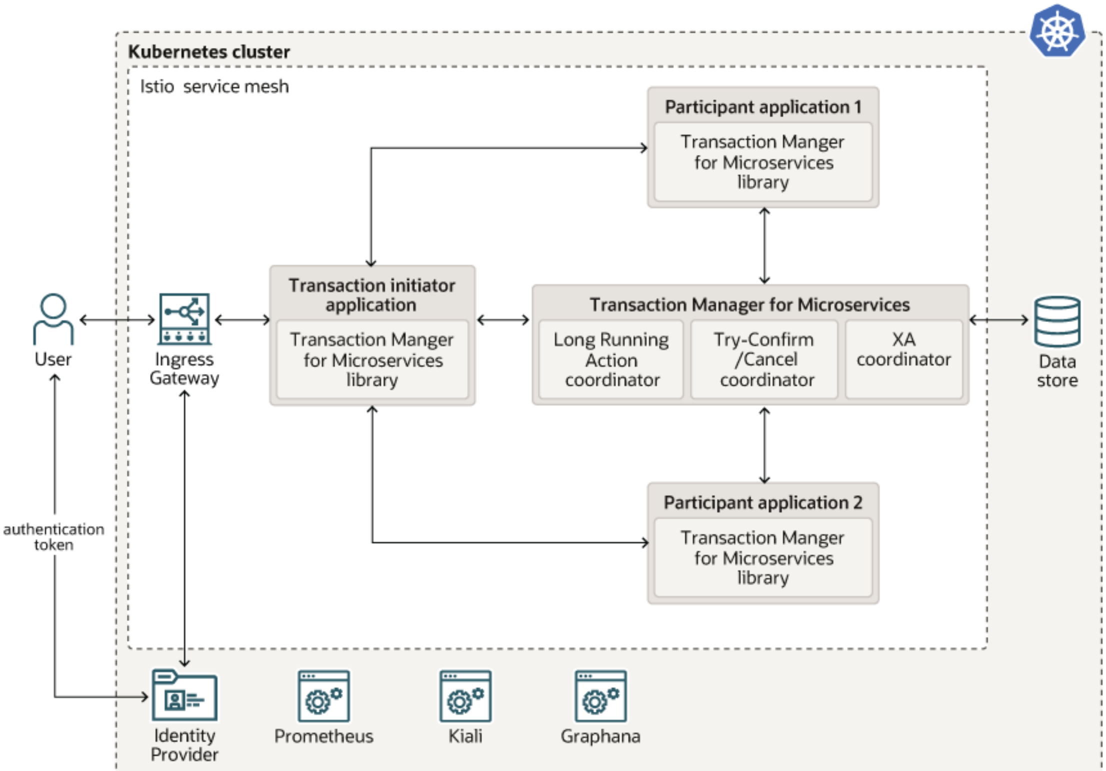
Install on a Kubernetes Cluster

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.

In Kubernetes, you can install MicroTx within a service mesh or without it. The installation bundle provides Helm charts and this section provides instructions to install MicroTx on a Kubernetes cluster with 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
Participant application 2
Transaction Manger for Microservices library
User
Ingress Gateway
Data store
authentication token
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.

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

Perform the following steps to install MicroTx:

1. Create a Kubernetes Cluster
2. Prepare the Environment
3. Create a Kubernetes Secret to Access Docker Registry
4. Push Images to a Remote Docker Repository
5. Authenticate and Authorize
6. Configure the values.yaml File
7. Install MicroTx

- Create a Kubernetes Cluster
  Create a Kubernetes cluster or use an existing one. You will install MicroTx onto this cluster.
- Prepare the Environment
  Before installing MicroTx, you must install the required software on your local machine and configure the environment on your local machine.
- 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.
- Push Images to a Remote Docker Repository
  The installation bundle that you have downloaded to your local system contains a Docker image of MicroTx.
- 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.
- Create a Kubernetes Secret for Oracle Database Credentials
  MicroTx supports using Oracle Database as a persistent store to keep track of the transaction information.
- 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.
- 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.
- 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 onto a Kubernetes cluster.
- 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.

- 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.

4.1 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.

4.2 Prepare the Environment

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

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 work with your Kubernetes cluster. See https://kubernetes.io/docs/tasks/tools/.

   Use `Kubectl` to create and manage your deployments. `Kubectl` uses the Kubernetes APIs to interact with the cluster.

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

   a. Run the following command to download Istio.

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

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

   ```
   cd istio-1.12.1
   ```

c. 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
```

d. 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!
```

e. Install Istio on the Kubernetes cluster with the default Istio profile. Oracle recommends using the default Istio profile for production environments. Additionally, enable distributed tracing and proxy access to logs in JSON format at the mesh level, using the following command:

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

This creates access logs which you can use for audit. 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/.

4. Create a namespace to deploy Transaction Manager for Microservices in the Kubernetes cluster. The following command creates a namespace with the name `otmm`, where `otmm` is the name of the namespace that you want to create:

Sample Command

```
kubectl create ns otmm
```

Sample Response

```
namespace/otmm created
```

5. 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
```

4.3 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.

4.4 Push Images to a Remote Docker Repository

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

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.

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 image of MicroTx 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 MicroTx image to the local Docker repository. The MicroTx image is located at `installation_directory/otmm-<version>/otmm/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. 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> <region-key>.ocir.io/otmmrepo/tmm:<version>
```

Where, `<region-key>.ocir.io/otmmrepo` is the remote 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 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:<version>
   ```

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.

4.5 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.

4.5.1 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"
   ```

4.5.2 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 password.

   The following command creates a Kubernetes secret with the name `TMMPRIVKEYPASSWD1` and key password as `Welcome1` in the `otmm` namespace.

   ```
   kubectl create secret generic TMMPRIVKEYPASSWD1 \ --from-
   literal=secret='Welcome1' -n otmm
   ```

   Where, `pwd...` is the private key password. Replace this with a value specific to your environment.

   > **Note:**
   >
   > Do not base64-encode the key password, as you must enter the key password 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.

4.6 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.

4.7 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.

4.8 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
   ```

4.9 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 the `values.yaml` file in any code editor. This file is located in the `*installation_directory*\otmm-*RELEASE*\otmm\helmcharts` folder. This file contains the 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 install MicroTx.
3. Save your changes.

- 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.
- Transaction Store Properties
  MicroTx uses a data store for persistence of transaction state.
- 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
  Under `authentication`, 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 `authTokenPropagationEnabled` under `tmmConfiguration.authorization`.
- Transaction Token Properties
  Under `transactionToken`, specify the key pair that you want to use for transaction token.

4.9.1 Environment Details

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

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`	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`
`applicationNameSpace`	Specify the namespace in which you want to deploy MicroTx. For example, `otmm`.
`tmmReplicaCount`	Enter **1** as you can only create a single replica of the MicroTx pod.

4.9.2 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`*.
`imagePullPolicy`	Enter `Always` to ensure that the image is pulled during the installation.

Property	Description
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.

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

4.9.3 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.
tmmid	Enter a value to uniquely identify each instance of MicroTx that you install. The unique identifier must have 5-characters 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 external URL to access MicroTx from outside the Kubernetes cluster where you have deployed the service. See Access MicroTx.
xaCoordinator, lraCoordinator, or tccCoordinator	Set enabled: "true" for the transaction protocols that you want to use. MicroTx supports three distribution transaction protocols: XA, LRA, and TCC. If you want to use nest an XA transaction within an LRA 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.

Property	Description
`narayanaLraCompatibilityMode`	Only for the LRA transaction protocol. Set `enabled` to `true` only when you want to use LRA participant applications that were implemented to work with the Narayana LRA Coordinator and now would participate in LRA transactions using MicroTx. Enable this mode to ensure that the MicroTx LRA 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`.
`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.

4.9.4 Transaction Store Properties

MicroTx uses a data 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 data 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 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. If you want to use multiple replicas of the transaction coordinator while using the internal memory as data store, you must enable session affinity.
`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 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.

Property	Description

`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. 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:7777/unique_connection_string_low.adb.oraclecloud.com?wallet_location=Database_Wallet`

`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.
`walletConfigMap.configMapName`	Enter the name of the configuration map that you have created for the wallet of the Autonomous Database instance. Example, `db-wallet-configmap`. You must provide a value for this field only if you are using an Autonomous Database instance. See Get Autonomous Database Client Credentials.

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 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`.
`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`.
`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`.
`cacertConfigMap.configMapFileName`	Enter the name of the PEM file that you had created while initializing the certificate authority. For example, `ca.pem`.

4.9.5 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.

Under `tmmConfiguration.authorization`, specify the details of the identity provider which you want to use for authorization.

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.
`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 `values.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.

4.9.6 Authentication Properties

Under `authentication`, 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.

Property	Description
`requestsWithNoJWT`	Enter `ALLOW` to bypass JWT authentication. This permits requests that do not have JWT tokens. Enter `DENY` 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. **Caution:** You must set this field to `DENY` in production environments.
`jwt.issuer`	Identifies the JWT token issuer.

Property	Description
`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 the `authentication` field in the `values.yaml` file. The sample values in this example are based on the values used in the sample commands in Run the Discovery URL.

```
authentication:
  requestsWithNoJWT: DENY
  jwt:
    issuer: "https://identity.oraclecloud.com"
    jwksUri: "https://idcs-
a83e4de370ea4db8c703a0b742ce74.identity.oraclecloud.com:443/admin/v1/
SigningCert/jwk"
```

4.9.7 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 key that you want to use for encrypting the transaction tokens.
`encryptionSecretKeys.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.
`encryptionSecretKeys.version`	Enter the version of the Kubernetes secrets that you want to use.

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

4.9.8 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"
```

4.10 Install MicroTx

Use Helm to install MicroTx onto 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.

   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
   ```

   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.

   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.

4.11 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.

4.12 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,

- *`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`.

5
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.
- 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.
- 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.
- 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.
- 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
- Run MicroTx in a Docker Container
  Additionally, you can use MicroTx in a separate Docker container. Follow the steps in this section to run MicroTx locally on a HTTPS port.

5.1 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/.

5.2 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/
   ```

5.3 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
   ```

5.4 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
```

5.5 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.

5.6 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
```

5.7 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
   ```

5.8 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.

5.8.1 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.
`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, LRA, and TCC. If you want to use nest an XA transaction within an LRA transaction, set `enabled: true` for both `xaCoordinator` and `lraCoordinator`.
`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.

Property	Description
`narayanaLraCompatibilityMode.enabled`	Only for the LRA transaction protocol. Set this property to `true` when you want to use LRA participant applications that were implemented to work with the Narayana LRA Coordinator and now would participate in LRA transactions using MicroTx. Enable this mode to ensure that the MicroTx LRA 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.

5.8.2 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`.
`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`.

Property	Description
`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`.

5.8.3 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
```

5.8.4 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.
`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.

5.8.5 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-
a83e4de370ea4db8c703a0b742ce74.identity.oraclecloud.com:443/admin/v1/
SigningCert/jwk"
```

5.8.6 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
```

5.8.7 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.
`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
```

5.9 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`: 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.

5.10 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.

5.11 Run MicroTx in a Docker Container

Additionally, you can use MicroTx in a separate Docker container. Follow the steps in this section to run MicroTx locally on a HTTPS port.

To run MicroTx along with a sample application on Docker Swarm, see Run Sample LRA Application in Docker Swarm.

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.

1. Place the `tcs.yaml` file in the current directory along with certificate and key files. If you have set `tmmConfiguration.serveTLS.enabled` to `true` in the `tcs.yaml` file to enable TLS, you must copy the certificate and key files into the current directory.
2. Run MicroTx using the configuration details provided in the `tcs.yaml` file.

   Sample Command

   ```
   docker container run --name otmm -v "$(pwd)":/app/config \
   -w /app/config -p 9000:9000/tcp --env CONFIG_FILE=tcs.yaml \
   --add-host host.docker.internal:host-gateway -d tmm:<version>
   ```

   Where,

   - `otmm` is the name of the container that you want to create.
   - `tmm:<`*`version`*`>` is the MicroTx Docker image that you have loaded to the local Docker repository.
3. After the installation is complete, you can access MicroTx. Run the following command to verify that you can access MicroTx.

   ```
   curl --cacert localhost.pem \
             -v -X POST  \
             -H "Content-Type: application/json" \
             https://localhost:9000/api/v1/xa-transaction
   ```

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

6

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 22.3.2
  MicroTx 22.3.2 provides additional features.
- Verify
  After installing MicroTx, run the following command to validate that the installation was completed successfully.
- Install MicroTx Library Files
  The MicroTx library for Java provides the functionality for your Java applications to initiate a new XA, LRA, or TCC transaction or to participate in an existing transaction. You must perform this task only once to install the library files on your system.

6.1 Upgrade to 22.3.2

MicroTx 22.3.2 provides additional features.

Skip this procedure if you have installed MicroTx 22.3.2.

Run these steps only if you have already installed MicroTx 22.3 or 22.3.1 and you want to avail the latest features in 22.3.2. For information about the new features, see Changes in MicroTx.

To upgrade to 22.3.2:

1. On https://www.oracle.com/database/transaction-manager-for-microservices/, click **Download MicroTx Free**, and then download the MicroTx installation bundle (.zip file).
2. Create a new directory in your local machine.
3. Extract the contents of the ZIP file to the new directory that you have created.

   ```
   unzip otmm-22.3.2.zip
   ```

   Ensure that you do not overwrite the installer files for earlier versions of MicroTx.
4. Run the following command to view the list of files that are extracted.

   ```
   ls -lR otmm-22.3.2
   ```

   This contains the updated image of the transaction coordinator at `*installation_directory*/otmm-22.3.2/otmm/image/tmm-22.3.2.tgz`. In the next steps, you will use this file to update the existing transaction coordinator image.
5. 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.

6. Update the YAML file for the transaction coordinator with the name of the latest image in the repo. 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.
7. 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 that you want to upgrade.
   - `otmm` is the namespace in Kubernetes cluster, where you have installed MicroTx.
   - *`installation_directory`*`/otmm-22.3.2/otmm/helmcharts/tmm` is the folder that contains the `chart.yaml` file for MicroTx.
   - *`installation_directory`*`/otmm-22.3.2/otmm/helmcharts/tmm/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.
8. Install the latest MicroTx client library files for Java. See Install MicroTx Library Files.

6.2 Verify

After installing MicroTx, run the following command to validate that the installation was completed successfully.

```
curl --location --request POST -H "Authorization:Bearer access_token"
https://externalHostname:externalPort/api/v1/xa-transaction
```

To identify values for the *externalHostname* and *externalPort*, see Access MicroTx. To create an access token, see Create an Access Token.

A HTTPS response with status 201 displays the internal and external URL to access MicroTx XA coordinator. A sample response is provided below.

```
{
    "internal": "http://otmm-tcs:9000/api/v1/xa-transaction/d369...",
    "external": "http://192.0.2.1/api/v1/xa-transaction/d3693..."
}
```

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

Some values have been truncated with ellipses (...) for readability in this example. When you run the command in your environment, you will see the entire response.

In the example response, you see the URL for XA coordinator as you have set `enabled: "true"` for the `xaCoordinator` field in the YAML file for MicroTx. If you enable the `lraCoordinator` or `tccCoordinator` fields, you will also get the URL for LRA and TCC coordinator.

6.3 Install MicroTx Library Files

The MicroTx library for Java provides the functionality for your Java applications to initiate a new XA, LRA, or TCC transaction or to participate in an existing transaction. You must perform this task only once to install the library files on your system.

Before you begin, ensure that you have installed Maven version 3.6 or later on your system. See https://maven.apache.org/download.cgi.

Run the following commands to install the MicroTx library files for Java. These files are available in the `installation_directory/otmm-<version>/lib/java` folder.

1. Install the `TmmLib-<version>.jar` file.

   ```
   mvn install:install-file \
          -Dfile=./lib/java/TmmLib-<version>.jar \
          -DgroupId=com.oracle.tmm.jta \
          -DartifactId=TmmLib \
          -Dversion=<version> \
          -Dpackaging=jar
   ```

2. Install the `TmmLib-<version>.pom` file.

   ```
   mvn install:install-file \
          -Dfile=./lib/java/TmmLib-<version>.pom \
          -DgroupId=com.oracle.tmm.jta \
          -DartifactId=TmmLib \
          -Dversion=<version> \
          -Dpackaging=pom
   ```

7 Deploy Sample Applications

Code for the sample applications is available in the Transaction Manager for Microservices (MicroTx) installation bundle. Using samples is the fastest way for you to get familiar with MicroTx.

The `installation_directory/otmm-RELEASE/samples` folder contains a sub-folder for each transaction protocol: XA, LRA, and TCC. Each sub-folder contains the sample application source code and files required by Helm.

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 using the MicroTx libraries with your application.

Deployment Details for Sample Microservices

Independently develop, test, and deploy the microservices. The applications must meet ACID requirements.

Deploy the sample microservices in the same namespace in which you have installed MicroTx.

Associate all the microservices with a single identity domain to share user definitions and authentication by using a common identity provider.

The MicroTx uses a data store to maintain data about global transactions and transaction logs.

For XA sample applications, the participant microservices connect to resource managers, which are external services for the participant microservices.

- Deploy XA Sample Application
  Let us understand how MicroTx manages transactions for applications that use the XA transaction protocol by using the sample XA application.
- Deploy LRA Sample Application
- Deploy TCC Sample Application

7.1 Deploy XA Sample Application

Let us understand how MicroTx manages transactions for applications that use the XA transaction protocol by using the sample XA application.

The XA sample application is available in the installation bundle in the `installation_directory/otmm-<version>/samples/xa` folder. This folder contains the code for three microservices, YAML files, and Helm charts for sample Java and Node.js sample applications. The sample application code is already configured to use the MicroTx libraries.

- Workflow to Run XA Sample Apps
  Use the following workflow as a guide to run the XA sample applications.

- 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).
- Identify a Sample App to Run
  The sample application code files are already updated to use the MicroTx client libraries. You can use these files as a reference when you are integrating MicroTx library code with your custom application.
- Set Up Resource Managers for Sample Apps
  Set up resource managers for Department One and Department Two in your sample XA application.
- Run Sample XA Application in Kubernetes
- Run Sample XA Application in Docker Swarm

7.1.1 Workflow to Run XA Sample Apps

Use the following workflow as a guide to run the XA sample applications.

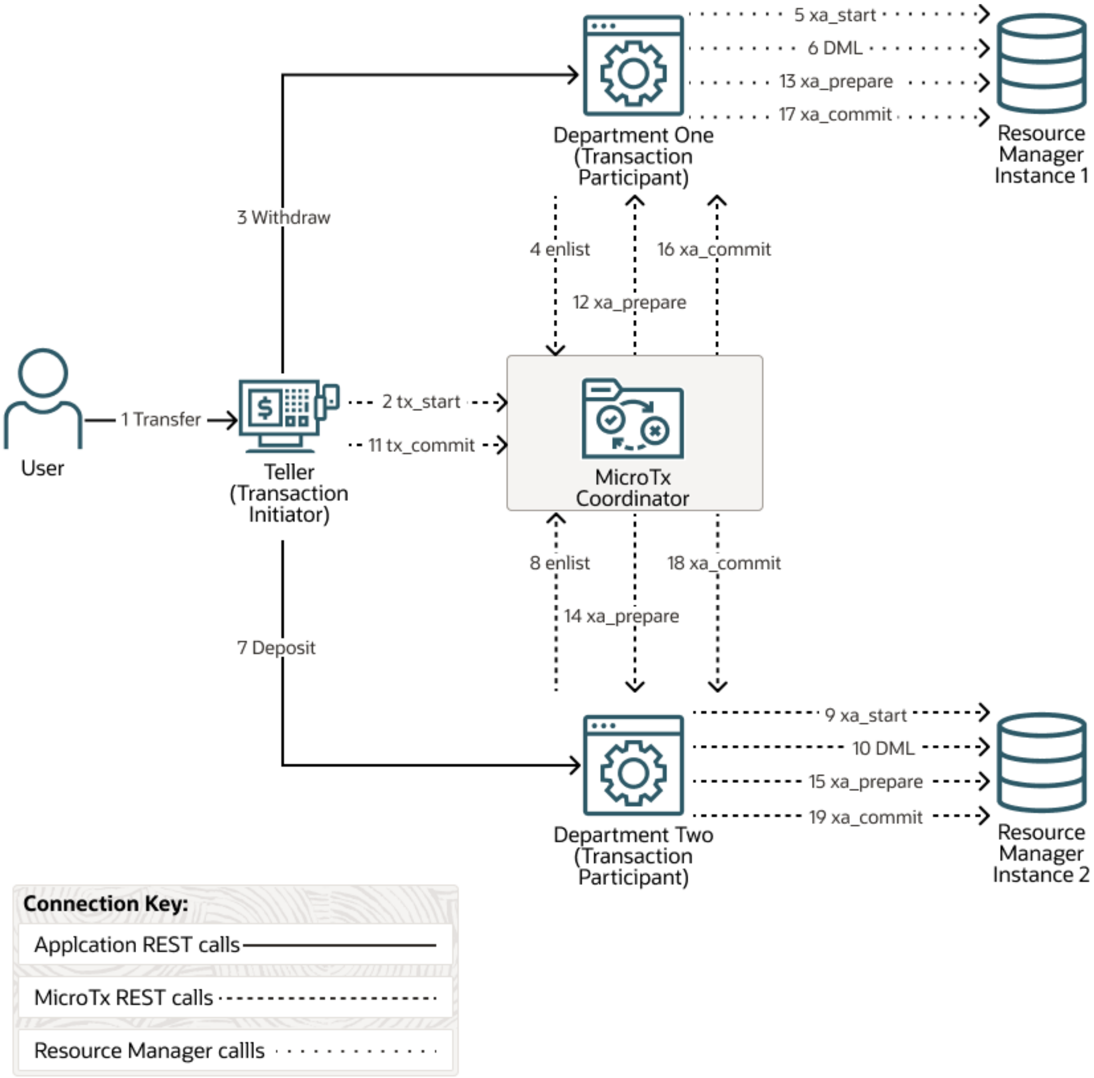
Task	Description	More Information
Create an access token	Download the installation bundle, set up a transaction store and identity provider.	Prepare
Install MicroTx	This guide provides instructions for you to install MicroTx in a Kubernetes cluster or Docker Swarm.	Install MicroTx in one of the following environments: • Install on a Kubernetes Cluster • Install on Docker Swarm
Learn about the components of the sample XA application	Sample applications are available for trying out different use cases. Identify the sample app that you want to run and note down the location of the source code for the sample application.	About XA Sample Application
Identify the XA sample application that you want to try out	Sample applications are available for trying out different use cases. Identify the sample app that you want to run and note down the location of the source code for the sample application.	Identify a Sample App to Run
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.	Set Up Resource Managers for Sample Apps
Build, install, and run the sample application	This guide provides instructions for you to run the sample applications in a Kubernetes cluster or Docker Swarm.	Run the sample apps in one of the following environments: • Run Sample XA Application in Kubernetes • Run Sample XA Application in Docker Swarm

7.1.2 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.

7.1.2.1 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 commnication 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.

7.1.3 Identify a Sample App to Run

The sample application code files are already updated to use the MicroTx client libraries. You can use these files as a reference when you are integrating MicroTx library code with your custom application.

The following table lists the combination of the XA sample applications that you can use to try out different scenarios. You'll need this information when you build images of the sample microservices and configure them. The table lists the relative path of the sample XA application code files within the `installation_directory/otmm-<version>/samples/xa` folder and `..` indicates this folder. Identify the scenario that you want to try out, and then note down the location of the source code for the sample application.

The Teller service is a transaction initiator service. Dept 1 and Dept 2 services are transaction participants services. You must set up a resource manager for all the transaction participant services. For more details, see About XA Sample Application.

Scenario	Location of Sample Code	Notes
Run Java sample applications using only XA-compliant resource managers. The Teller service only initiates the transaction and does not participate in it, so it does not require a resource manager.	Initiator app: `../java/teller` Dept 1: `../java/department-helidon` Dept 2: `../java/department-spring`	Set up XA-compliant resource managers for Dept 1 and Dept 2. See Set Up XA-Compliant Resource Manager.

Scenario	Location of Sample Code	Notes
Run Java sample applications using only XA-compliant resource managers. The Teller service initiates, and then participates in the transaction, so it also requires a resource manager.	Initiator app: `../java/teller-as-participant` Dept 1: `../java/department-helidon` Dept 2: `../java/department-spring`	Set up XA-compliant resource managers for Teller, Dept 1, and Dept 2 services. See Set Up XA-Compliant Resource Manager.
Run Node.js sample applications using only XA-compliant resource managers. The Teller app only initiates the transaction and does not participate in it, so it does not require a resource manager.	Initiator app: `../nodejs/teller` Dept 1: `../java/department-helidon` Dept 2: `../nodejs/department`	Set up XA-compliant resource managers for Dept 1 and Dept 2. See Set Up XA-Compliant Resource Manager.
Run Java sample applications using an XA-compliant resource manager for Dept 1 and a resource that does not support XA and JDBC for Dept 2. The Teller service only initiates the transaction, so it does not require a resource manager.	Initiator app: `../java/teller` Dept 1: `../java/department-helidon` Dept 2: `../java/department-nonxa-ds`	MySQL is a JDBC resource which is not XA-compliant. Use Logging Last Resource (LLR) optimization to enable MySQL to participate in a distributed transaction. Set up MySQL as a resource manager for Dept 2. See Set Up MySQL for Sample Participant Services.
Run Java sample applications using an XA-compliant resource manager for Dept 1 and MongoDB or PostgreSQL as resource manager for Dept 2. The Teller service only initiates the transaction, so it does not require a resource manager.	Initiator app: `../java/teller` Dept 1: `../java/department-helidon` Dept 2: `../java/department-nonxa`	Mongo DB is a non-XA and non-JDBC resource. Use LLR optimization to enable MongoDB to participate in a distributed transaction. See Set Up MySQL for Sample Participant Services. PostgreSQL is an XA-compliant resource. To use PostgreSQL as a resource manager, you must make additional changes to your application code. See Configure PostgreSQL as Resource Manager.
Run Java sample applications using XA-compliant resource managers for Dept 1 and Dept 2 services. The Teller service initiates, and then participates in the transaction, so you must set up a resource manager for the Teller service.	Initiator app: `../java/teller-as-participant-nonxa-ds` Dept 1: `../java/department-helidon` Dept 2: `../java/department-spring`	MySQL is a JDBC resource which is not XA-compliant. Use LLR optimization to enable MySQL to participate in a distributed transaction. Set up MySQL as a resource manager for the Teller service. See Set Up MySQL for Teller Service.
Run Java sample applications using an XA-compliant resource manager for Dept 1 and MySQL as resource manager for Dept 2. The Teller service only initiates the transaction, so it does not require a resource manager.	Initiator app: `../java/teller` Dept 1: `../java/department-helidon` Dept 2: `../java/department-nonxa-lrc-ds`	MySQL is a JDBC resource which is not XA-compliant. Use Last Resource Commit (LRC) optimization to enable MySQL to participate in a distributed transaction. Set up MySQL as a resource manager for Dept 2. See Set Up MySQL for Sample Participant Services.

Scenario	Location of Sample Code	Notes
Run Java sample applications using an XA-compliant resource manager for Dept 1 and MongoDB as resource manager for Dept 2. The Teller service only initiates the transaction, so it does not require a resource manager.	Initiator app: `../java/teller` Dept 1: `../java/department-helidon` Dept 2: `../java/department-nonxa-lrc`	Mongo DB is a non-XA and non-JDBC resource. Use LRC optimization to enable MongoDB to participate in a distributed transaction. See Set Up MongoDB as Resource Manager.

7.1.4 Set Up Resource Managers for Sample Apps

Set up resource managers for Department One and Department Two in your sample XA application.

For Department One, you can use any XA-compliant database as a resource manager. For example, Autonomous Transaction Processing (ATP) Database instances in Oracle Cloud.

For Department Two, set up one of the following as a resource manager based on the use case that you want to implement:

- XA-compliant database
- Non-XA compliant data stores, such as MongoDB and MySQL
- PostgreSQL as database

In the use case where the Teller service initiates and then participates in a transaction, you must set up a resource manager for the Teller service.

- Set Up XA-Compliant Resource Manager
  Set up XA-compliant resource managers for your sample XA application, and then create tables with sample values.
- Set Up MongoDB as Resource Manager
  MicroTx supports MongoDB 4.1 or later as a resource manager. Mongo DB is a non-XA and non-JDBC resource. Use Logging Last Resource (LLR) or Last Resource Commit (LRC) optimization to enable MongoDB to participate in a distributed transaction.
- Set Up MySQL for Teller Service
  MySQL is a JDBC resource which is not XA-compliant. Use Logging Last Resource (LLR) or Last Resource Commit (LRC) optimization to enable MySQL to participate in a distributed transaction as the data store for the Teller service, the transaction initiator service.
- Set Up MySQL for Sample Participant Services
  MySQL is a JDBC resource which is not XA-compliant. Use Logging Last Resource (LLR) or Last Resource Commit (LRC) optimization to enable MySQL to participate in a distributed transaction.
- Configure PostgreSQL as Resource Manager
  To use PostgreSQL as resource manager for Dept 2 in the XA sample application, you must update a few YAML files and enable session affinity.
- Enable Session Affinity for XA Participants
  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.

7.1.4.1 Set Up XA-Compliant Resource Manager

Set up XA-compliant resource managers for your sample XA application, and then create tables with sample values.

You can use any Oracle Database. For example, Autonomous Transaction Processing (ATP) Database instances in Oracle Cloud, an Oracle Database running inside a Kubernetes cluster, or an on-premises database. Ensure that MicroTx and the application, when it is deployed, can access the database.

Only if you use 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.

3. Copy the wallet files to the following folders that contain the source code for the participant applications.

   - `*installation_directory*/otmm-*RELEASE*/samples/xa/java/department-helidon/Database_Wallet`
   - `*installation_directory*/otmm-*RELEASE*/samples/xa/java/department-spring/Database_Wallet`

Create database and tables with sample values

To test the sample XA applications, create database and tables with sample values for both the department applications. The MicroTx installation bundle includes the SQL script file that you can run to create the required tables. Run the SQL script using a client tool with which you connect to the database. You'll need to provide database credentials to establish a connection with the database and run the SQL script.

To use the SQL script to create a database, a table, and populate it with sample values:

1. Run the `<*installation_directory*/otmm-*RELEASE*/samples/xa/java/department-helidon/department.sql` file by connecting to Oracle Database by using SQL developer or SQL plus.
   This creates a database with the name `department_helidon` and a table with the name `accounts`. It also populates the `accounts` table with sample values.

2. Run the `<*installation_directory*/otmm-*RELEASE*/samples/xa/java/department-spring/department.sql` file by connecting to Oracle Database by using SQL developer or SQL plus.
   This creates a database with the name `department_spring` and a table with the name `accounts`. It also populates the `accounts` table with sample values as provided in the following table.

Account_ID	Amount
account1	1000
account2	2000
account3	3000
account4	4000
account5	5000

7.1.4.2 Set Up MongoDB as Resource Manager

MicroTx supports MongoDB 4.1 or later as a resource manager. Mongo DB is a non-XA and non-JDBC resource. Use Logging Last Resource (LLR) or Last Resource Commit (LRC) optimization to enable MongoDB to participate in a distributed transaction.

1. Set up MongoDB with transaction capability. To support transactions, you must set up MongoDB replication. See https://www.mongodb.com/docs/manual/replication/#transactions.
2. Run the following commands to create a table in MongoDB with seed data for the sample XA application.

   ```
   db.createCollection("accounts")
   db.accounts.insertMany([{"accountId":"111", "name":"account1",
   "amount":1000.00},{"accountId":"222", "name":"account2",
   "amount":2000.00},{"accountId":"333", "name":"account3",
   "amount":3000.00}])
   ```

3. Create the `commitRecords` collection for storing the committed records.

   ```
   db.createCollection("commitRecords")
   ```

   Skip this step if you are running the sample app for Last Resource Commit (LRC) optimization.
4. Enable session affinity or sticky sessions for the participant service. See Enable Session Affinity for XA Participants.

7.1.4.3 Set Up MySQL for Teller Service

MySQL is a JDBC resource which is not XA-compliant. Use Logging Last Resource (LLR) or Last Resource Commit (LRC) optimization to enable MySQL to participate in a distributed transaction as the data store for the Teller service, the transaction initiator service.

In this scenario, the transaction initiator and participant applications are Java applications. Use XA-compliant resource managers for Dept 1 and Dept 2 application. Set up MySQL as resource manager for the Teller application.

Set up a resource manager for the Teller application only when you want to try out the scenario where you use an initiator application as a participant as well. The banking teller application transfers an amount from one department to another. For every transaction, the teller application charges an amount 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.

Perform the following steps to set up a resource manager for the Teller application:

1. Set up MySQL. For information about installation and configuration, refer to the MySQL documentation.
2. Run the following sample commands to create a table in MySQL with seed data for the sample XA application. Use the `fee` table to demonstrate the commission charged by the Teller application.

   ```
   create database transfer_fee;
   use transfer_fee;
   create table fee
   (
       account_id VARCHAR(10) not null,
       amount decimal(10,2) not null,
       PRIMARY KEY (account_id)
   );
   insert into fee values('account1', 10.00);
   insert into fee values('account2', 20.00);
   insert into fee values('account3', 30.00);
   insert into fee values('account4', 40.00);
   insert into fee values('account5', 50.00);
   ```

3. Create a table for storing the committed records.

   ```
   CREATE TABLE LLR_COMMIT_RECORD (
       GTRID varchar(255) NOT NULL,
       DATE_COMMITED TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
       RECORDSTR text,
       PRIMARY KEY (GTRID)
   ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
   ```

   Skip this step if you are running the sample app for Last Resource Commit (LRC) optimization.

4. Enable session affinity or sticky sessions for the participant service. See Enable Session Affinity for XA Participants.

7.1.4.4 Set Up MySQL for Sample Participant Services

MySQL is a JDBC resource which is not XA-compliant. Use Logging Last Resource (LLR) or Last Resource Commit (LRC) optimization to enable MySQL to participate in a distributed transaction.

1. Set up MySQL. For information about installation and configuration, refer to the MySQL documentation.
2. Run the following commands to create a table in MySQL with seed data for the sample XA application.

   ```
   create database department_nonxa_ds;
   use department_nonxa_ds;
   create table accounts
   (
       account_id VARCHAR(10) not null,
       name VARCHAR(60) not null,
   ```

```
    amount decimal(10,2) not null,
    PRIMARY KEY (account_id)
);
insert into accounts values('account1', 'account1', 1000.00);
insert into accounts values('account2', 'account2', 2000.00);
insert into accounts values('account3', 'account3', 3000.00);
insert into accounts values('account4', 'account4', 4000.00);
insert into accounts values('account5', 'account5', 5000.00);
```

3. Create a table for storing the committed records.

   ```
   CREATE TABLE LLR_COMMIT_RECORD (
       GTRID varchar(255) NOT NULL,
       DATE_COMMITED TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
       RECORDSTR text,
       PRIMARY KEY (GTRID)
   ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
   ```

   Skip this step if you are running the sample app for Last Resource Commit (LRC) optimization.

4. Enable session affinity or sticky sessions for the participant service. See Enable Session Affinity for XA Participants.

7.1.4.5 Configure PostgreSQL as Resource Manager

To use PostgreSQL as resource manager for Dept 2 in the XA sample application, you must update a few YAML files and enable session affinity.

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

To configure PostgreSQL as a resource manager:

1. In any code editor, open the `Configuration.java` file located in the *`installation_directory`*`/otmm-`*`RELEASE`*`/samples/xa/java/department-spring/src/main/java/com/oracle/mtm/sample` folder.
2. Remove comments from all the lines of code which have the following comment: `Uncomment when the application uses PostgreSQL`.
3. In any code editor, open the `pom.xml` file located in the *`installation_directory`*`/otmm-`*`RELEASE`*`/samples/xa/java/department-spring` folder.
4. Remove comments from all the lines of code which have the following comment: `Uncomment when the application uses PostgreSQL`, so that the following details about the driver are no longer commented.

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

5. 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 for XA Participants.

7.1.4.6 Enable Session Affinity for XA Participants

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.

You must enable session affinity or sticky sessions for the transaction participant service in the following scenarios:

- When the participant service uses PostgreSQL as a resource manager.
- When the participant service uses a non-XA resource as a resource manager.

To enable session affinity for a participant service:

1. Create a networking rule for the participant service in the namespace where you have deployed the service. The traffic policy must use a load balancer with consistent hash that uses the HTTP request header, `oracle-tmm-txn-id`.
2. Update the `networking.yaml` file located in the *installation_directory*/otmm-*RELEASE*/samples/xa/helmcharts/sampleapps/templates folder. Specify the `oracle-tmm-txn-id` HTTP header in Istio's `DestinationRule` resource to enable session affinity or create sticky sessions. 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
   spec:
       host: {{$val.host}}
       trafficPolicy:
         loadBalancer:
           consistentHash:
             httpHeaderName: oracle-tmm-txn-id
   ```

   Where,

   - *host*: Specify the fully qualified name of your application inside the Kubernetes cluster. For example, `dept1.otmm.svc.cluster.local`.
3. In any code editor, open the `values.yaml` file located in the *installation_directory*/otmm-*RELEASE*/samples/xa/helmcharts/sampleapps folder.
4. In the `values.yaml` file, under `Dept2`, search for the `sessionAffinity` property and set the value to `true`.

   ```
   sessionAffinity: true
   ```

7.1.5 Run Sample XA Application in Kubernetes

- Build Docker Images for Sample XA Application
  Before you begin building the Docker images, ensure that you have completed the following tasks.
- Push XA Sample App Images
  Push the Docker image of the sample applications, that you have built, to a remote repository.
- Update the values.yaml File for XA Sample App
  The sample application files also contain the `values.yaml` file, the manifest file of the sample application, which contains the deployment configuration details for the XA sample application.
- Install XA Sample Application
  Install the XA sample application in the Kubernetes cluster where you have installed MicroTx.
- Run an XA Transaction

7.1.5.1 Build Docker Images for Sample XA Application

Before you begin building the Docker images, ensure that you have completed the following tasks.

1. Installed MicroTx.
2. Identified the sample application that you want to try out and noted down the location of the code files. See Identify a Sample App to Run.
3. Set up resource managers for participant services. Copied the wallet files to the sample application folders if you are using an Autonomous Database instances as resource manager.

The code samples in the following procedure use the location of a Java sample application which uses XA-compliant resource managers. Update the path to the sample application code file based on the scenario that you want to try out.

Perform the following steps to build Docker images for each microservice in the sample:

1. Run the following commands to build the Docker image for the Teller application.

   Sample command

   ```
   cd installation_directory/otmm-<version>/samples/xa/java/teller
   docker image build -t teller:1.0 .
   ```

2. Run the following commands to build the Docker image for the Department 1 application.

   Sample command

   ```
   cd installation_directory/otmm-<version>/samples/xa/java/department-helidon
   docker image build -t department-helidon:1.0 .
   ```

3. Run the following commands to build the Docker image for the Department 2 application.

   Sample command

   ```
   cd installation_directory/otmm-<version>/samples/xa/java/department-spring
   docker image build -t department-spring:1.0 .
   ```

The Docker images that you have created are available in your local Docker container registry. Note down the names of the images as you will provide this information in the next step.

7.1.5.2 Push XA Sample App Images

Push the Docker image of the sample 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 the sample application.

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. 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.

   Sample commands

   The following sample commands tag the images of the Teller, department 1, and department 2 XA applications. Provide the image names based on your environment.

   Sample Code

   ```
   docker tag teller:1.0 <region-key>.ocir.io/otmmrepo/teller:1.0
   docker tag dept1:1.0 <region-key>.ocir.io/otmmrepo/dept1:1.0
   docker tag dept2:1.0 <region-key>.ocir.io/otmmrepo/dept2:1.0
   ```

   Where, `<region-key>.ocir.io/otmmrepo` is the Oracle Cloud Infrastructure Registry to which you want to push the image file. If you are using other Kubernetes platforms, then provide the details based on your environment.
3. Push the Docker image from your local repository to the remote Docker repository.

   Syntax

   ```
   docker push remote_image[:tag]
   ```

   Sample commands

The the following sample commands push the tagged images of Teller, department 1, and department 2 applications. Provide the image names based on your environment.

Sample Code

```
docker push <region-key>.ocir.io/otmmrepo/teller:1.0
docker push <region-key>.ocir.io/otmmrepo/dept1:1.0
docker push <region-key>.ocir.io/otmmrepo/dept2:1.0
```

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.

7.1.5.3 Update the values.yaml File for XA Sample App

The sample application files also contain the `values.yaml` file, the manifest file of the sample application, which contains the deployment configuration details for the XA sample application.

While deploying the sample application to a Kubernetes cluster, Helm pulls the sample application images from the remote Docker registry. In the `values.yaml` file of the sample application, specify the image to pull and the credentials to use when pulling the images. Also provide details to access the resource managers.

To provide the configuration and environment details in the `values.yaml` file:

1. Open the `values.yaml` file, which is located in the `installation_directory/otmm-<version>/samples/xa/java/helmcharts/transfer` folder, in any code editor.

   This file contains sample values. Use this file as a reference to create your own YAML file to run and manage microservices in Kubernetes.

2. Provide details to access the resource manager for Department 1 and Department 2 microservices.

   - `connectString`: Enter the public URL to access the database. It can be the public IP address of the database node or the IP address of the cluster.

     – 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
```

- `databaseUser`: Enter the user name to access the database, such as `SYS`.
- `databasePassword`: Enter the password to access the database for the specific user.

3. Provide details of all the sample application images that you have uploaded to the docker container. For example, `iad.ocir.io/mytenancy/xa/teller:1.0`.
4. Save your changes.

7.1.5.4 Install XA Sample Application

Install the XA sample application in the Kubernetes cluster where you have installed MicroTx.

1. Navigate to the folder that contains the Helm Charts. Provide the path for the sample application that you want to try out.

   Sample Code

   ```
   cd installation_directory/otmm-<version>/otmm/samples/xa/helmcharts
   ```

2. Run the following command to install the XA sample application.

   ```
   helm install sample-xa-app --namespace otmm transfer/ \
   --values transfer/values.yaml
   ```

   Where, *sample-xa-app* is the name of the application that is installed.

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
   ```

7.1.5.5 Run an XA Transaction

Before you start a transaction, you must create an access token, install the MicroTx library files, and note down the external IP address of the Istio ingress gateway.

1. Before starting the transaction, run the following commands to check the balance in department 1 and department 2.

   ```
   curl --location --request GET -H "Authorization:Bearer $TOKEN"
   'http://$CLUSTER_IPADDR/dept1/account1' | jq
   curl --location --request GET -H "Authorization:Bearer $TOKEN"
   'http://$CLUSTER_IPADDR/dept2/account2' | jq
   ```

   Where,

   - *CLUSTER_IPADDR* is the name of the variable in which you stored the external IP address of the Istio ingress gateway. For information about finding the external IP address of the Istio ingress gateway and storing it in a variable, see Find IP Address of Istio Ingress Gateway.
   - *TOKEN* is the name of the variable in which you stored the authentication token earlier. For information about retrieving the authentication token and storing it in a variable, see Create an Access Token. You don't have to create and specify an authentication token only if your test environment is a Minikube cluster in which you perform the operations in a single cluster that's available on your local machine.

2. Transfer an amount of 50 from department 1 to department 2.

   ```
   curl --location --request POST -H "Authorization:Bearer $TOKEN"
   'http://$CLUSTER_IPADDR/transfers' --header 'Content-Type: application/
   json' --data-raw '{"from" : "account1", "to" : "account2", "amount" : 50}'
   ```

   Based on the business logic, 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. MicroTx prepares the participant services, Department One and Department Two, and then commits the transactions.

3. Check balances in department 1 and department 2 to verify that the amounts reflect correctly after the transaction. Run the following commands to confirm the transaction.

   ```
   curl --location --request GET -H "Authorization:Bearer $TOKEN"
   'http://$CLUSTER_IPADDR/dept1/account1' | jq
   curl --location --request GET -H "Authorization:Bearer $TOKEN"
   'http://$CLUSTER_IPADDR/dept2/account2' | jq
   ```

4. Run the following command to check the balance in department 1, and note down the balance. You will compare the account balance after a few steps.

   ```
   curl --location --request GET -H "Authorization:Bearer $TOKEN"
   'http://$CLUSTER_IPADDR/dept1/account1' | jq
   ```

5. To test how MicroTx handles failures and performs rollback, transfer an amount of 100 from department 1 to department 2, but specify an account number which does not exist, such as `account10`.

   ```
   curl --location --request POST -H "Authorization:Bearer $TOKEN"
   'http://$CLUSTER_IPADDR/transfers' --header 'Content-Type: application/
   json' --data-raw '{"from" : "account1", "to" : "account10", "amount" :
   100}'
   ```

You will receive the `500 Internal server` error. The transaction participant service will receive an error message that `account10` does not exist.

In case of any failure, the Teller service calls rollback instead of commit.

6. Run the following command to check the balance in department 1.

   ```
   curl --location --request GET -H "Authorization:Bearer $TOKEN"
   'http://$CLUSTER_IPADDR/dept1/account1' | jq
   ```

   Check if there is any change in the balance. If the balance remains the same, it indicates that the amount was not withdrawn from department 1.

7.1.6 Run Sample XA Application in Docker Swarm

- Build and Push the Docker Images
  Before you begin building the Docker images, ensure that you have copied the wallet files to the sample application folders if you are using an Autonomous Database instances as resource manager.
- Install XA Sample Application
  Install the XA sample application in Docker Swarm.
- Run an XA Transaction
  Before you start a transaction, you must install the Transaction Manager for Microservices library files and push the Docker image of the services to the Docker registry.

7.1.6.1 Build and Push the Docker Images

Before you begin building the Docker images, ensure that you have copied the wallet files to the sample application folders if you are using an Autonomous Database instances as resource manager.

1. Installed MicroTx.
2. Identified the sample application that you want to try out and noted down the location of the code files. See Identify a Sample App to Run.
3. Set up resource managers for participant services. Copied the wallet files to the sample application folders if you are using an Autonomous Database instances as resource manager.

It is important that you tag the Docker images that you build with the address of the registry that you have created. For example, *`192.0.2.1:5000`*. This is required while distributing the apps to the Swarm.

The code samples in the following procedure use the location of a Java sample application which uses XA-compliant resource managers. Update the path to the sample application code file based on the scenario that you want to try out.

Perform the following steps to build Docker images for each microservice in the sample:

1. Store the location of the Docker registry in an environment variable named `REGISTRY_LOCATION` as shown in the following command.

   ```
   export REGISTRY_LOCATION=192.0.2.1:5000
   ```

   Where,

   - `192.0.2.1` is the IP address of the Docker registry that you have created.
   - `5000` is the port number over which the Docker registry container communicates. Ensure that you have set up the required networking rules to permit inbound and outbound HTTPS or HTTP traffic over this port.

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

2. Run the following commands to build the Docker image for the Teller application.

   Sample command

   ```
   cd installation_directory/otmm-<version>/samples/xa/java/teller
   docker image build -t $REGISTRY_LOCATION/teller:1.0 .
   ```

3. Run the following commands to build the Docker image for the Department 1 application.

   Sample command

   ```
   cd installation_directory/otmm-<version>/samples/xa/java/department-helidon
   docker image build -t $REGISTRY_LOCATION/department-helidon:1.0 .
   ```

4. Run the following commands to build the Docker image for the Department 2 application.

   Sample command

   ```
   cd installation_directory/otmm-<version>/samples/xa/java/department-spring
   docker image build -t $REGISTRY_LOCATION/department-spring:1.0 .
   ```

5. Push the tagged Docker image to the Docker registry that you have created.

   Syntax

   ```
   docker push image[:tag]
   ```

   Sample commands

   The the following sample commands push the tagged images of hotel, flight, and trip manager applications.

   ```
   docker push $REGISTRY_LOCATION/teller:1.0
   docker push $REGISTRY_LOCATION/department-helidon:1.0
   docker push $REGISTRY_LOCATION/department-spring:1.0
   ```

   When you build the Docker images, they are available in your local Docker container registry. When you push the Docker image, it becomes available in the Docker registry that you have created for the Swarm.

7.1.6.2 Install XA Sample Application

Install the XA sample application in Docker Swarm.

All Swarm objects are described in manifests called *stack files*. The `tmm-stack-compose.yaml` stack file is located at `installation_directory/otmm-<version>/samples/docker`. This is a sample YAML file which describes all the components and configurations of the XA sample application and transaction coordinator. Use this file as a reference to create your own YAML file to run and manage microservices in Docker Swarm.

To run XA sample application:

1. Edit the `tmm-stack-compose.yaml` stack file in any code editor.

   This file contains the configuration details for the XA sample applications. Uncomment the section for XA sample applications.

2. Provide details to access the resource manager for Department 1 and Department 2 microservices.

   - `DEPARTMENTDATASOURCE_URL`: Enter the public URL to access the resource manager. It can be the public IP address of the database node or the IP address of the cluster.
   - `DEPARTMENTDATASOURCE_USER`: Enter the user name to access the resource manager, such as `SYS`.
   - `DEPARTMENTDATASOURCE_PASSWORD`: Enter the password to access the resource manager for the specificied user.

   For information about identifying values for these fields, see Distributed Transactions in *JDBC Developer's Guide and Reference*.

3. Provide details of all the sample application images that you have uploaded to the Docker registry. For example, `198.51.100.1:5000/teller:1.0`.

4. Save your changes.

5. Deploy the `tmm-stack-compose.yaml` stack file.

   ```
   cd installation_directory/otmm-<version>/samples/docker
   docker stack deploy -c tmm-stack-compose.yaml tmmdemo
   ```

   Where, `tmmdemo` is the name of the Docker stack that you want to install. You can specify any other name.

   ```
   Output:
   Creating network tmmdemo_default
   Creating config tmmdemo_my_tcs_config
   Creating service tmmdemo_dept1
   Creating service tmmdemo_dept2
   Creating service tmmdemo_teller
   Creating service tmmdemo_otmm-tcs
   ```

6. Verify that all services are ready. Use the following command to retrieve the list of services and their status.

   ```
   docker service ls
   ```

   The following sample output shows that all the services are ready.

   ```
   ID             NAME                   MODE         REPLICAS
   IMAGE                                    PORTS
   tjc0u55yavu4   registry               replicated   1/1
   registry:2                               *:5000->5000/tcp
   varg9g3astj4   tmmdemo_dept1          replicated   1/1
   198.51.100.1:5000/department-helidon:1.0   *:8086->8080/tcp
   ovtkx3677ypa   tmmdemo_dept2          replicated   1/1
   198.51.100.1:5000/department-spring:1.0    *:8087->8082/tcp
   ilkvx4emyv8c   tmmdemo_otmm-tcs       replicated   1/1
   198.51.100.1:5000/tmm:latest               *:9000->9000/tcp
   jv80wxsehbd2   tmmdemo_teller         replicated   1/1
   198.51.100.1:5000/teller:1.0               *:8085->8080/tcp
   ```

   Note down the port numbers on which the applications are running as you will need to provide the port number when you run the sample application.

When the services are ready, you can run an XA transaction.

7.1.6.3 Run an XA Transaction

Before you start a transaction, you must install the Transaction Manager for Microservices library files and push the Docker image of the services to the Docker registry.

1. Before starting the transaction, run the following commands to check the balance in Department 1 and Department 2.

   ```
   curl --location --request GET http://$REGISTRY_IPADDR:8086/accounts/
   account1 | jq
   curl --location --request GET http://$REGISTRY_IPADDR:8087/accounts/
   account2 | jq
   ```

   Where,

   - *REGISTRY_IPADDR* is the name of the variable in which you stored the IP address of the Docker registry to which you have pushed the Docker images. For information about storing the IP address of the Docker registry in a variable, see Access MicroTx in Docker Swarm.
   - 8086 and 8087 are the port numbers on which the Department 1 and Department 2 services are running respectively.

   Provide these details based on your environment.

2. Transfer an amount of 50 from Department 1 to Department 2.

   ```
   curl --location --request POST http://$REGISTRY_IPADDR:8085/
   transfers --header 'Content-Type: application/json' --data-raw
   '{"from" : "account1", "to" : "account2", "amount" : 50}'
   ```

   Where, `8085` is the port number on which the Teller service is running. Provide the port number information based on your environment.

3. Check the balance in Department 1 and Department 2 to verify that the account balance are updated correctly after the transaction. Run the following commands to confirm the transaction.

   ```
   curl --location --request GET http://$REGISTRY_IPADDR:8086/accounts/
   account1 | jq
   curl --location --request GET http://$REGISTRY_IPADDR:8087/accounts/
   account2 | jq
   ```

7.2 Deploy LRA Sample Application

- About the Sample LRA Application
  The LRA sample application is available in the installation bundle in the `installation_directory/otmm-<version>/samples/lra/lrademo` folder. This folder contains the code for three sample microservices, YAML files, and Helm charts.
- Run Sample LRA Application in Kubernetes
- Run Sample LRA Application in Docker Swarm

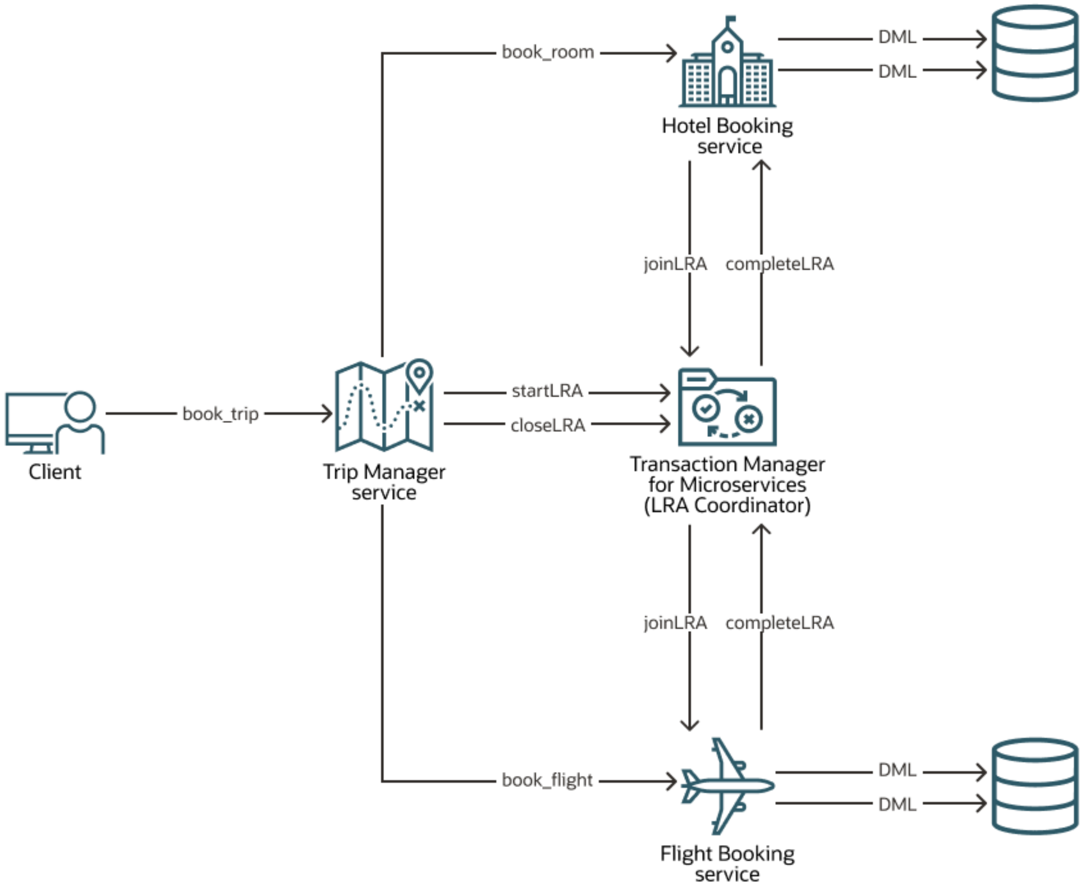
7.2.1 About the Sample LRA Application

The LRA sample application is available in the installation bundle in the `installation_directory/otmm-<version>/samples/lra/lrademo` folder. This folder contains the code for three sample microservices, YAML files, and Helm charts.

Microservices in Sample LRA Application

The following figure shows a sample LRA application, which contains several microservices, to demonstrate how you can use MicroTx to manage LRA transactions.

Use the sample application to book a trip, which consists of booking a hotel room and a flight. 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 LRA application consists of the following polyglot microservices:

- MicroTx (LRA Coordinator) coordinates the transaction between the sample microservices.
- Trip Manager service is the transaction initiator service, where the LRA 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. This Java application is located at `installation_directory/otmm-<version>/samples/lra/lrademo/trip-manager`.
- 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 LRA, it enlists itself and provides callback URIs that the LRA coordinator uses to complete or compensate the room reservation. This Java application is located at `installation_directory/otmm-<version>/samples/lra/lrademo/hotel`.
- 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 LRA, it enlists itself and provides callback URIs that the LRA coordinator uses to complete or compensate the flight reservation. This TypeScript application is located at `installation_directory/otmm-<version>/samples/lra/lrademo/flight`.
- Trip client is the user interface which you can use to confirm or cancel the booking. It does not participate in the LRA transaction. It is provided as sample client service which calls microservices to perform a distributed transaction that uses the LRA protocol. This

Java application is located at `installation_directory/otmm-<version>/samples/lra/lrademo/trip-client`.

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.

- Scenario: Book a Seat in a Cinema
  MicroTx supports a single level of nesting. You can only nest an XA transaction within an LRA transaction.

7.2.1.1 Scenario: Book a Seat in a Cinema

MicroTx supports a single level of nesting. You can only nest an XA transaction within an LRA transaction.

Let's understand how XA transactions are nested within an LRA transaction by using a sample application that books a cinema ticket. This sample application is not available in the installation bundle.

Microservices in the Sample Nested Transaction

The sample application which nests an XA transaction within an LRA transaction consists of the following polyglot microservices:

- MicroTx (LRA coordinator and XA coordinator)
- Seat booking service: This is the transaction initiator service, where the LRA transaction starts. This service reserves a seat and calls the payment service to handle the payment for the reserved seat.
- Payment service: This service is called within the context of an existing LRA, it enlists itself and provides callback URIs that the LRA coordinator uses to complete or compensate the seat reservation.
  It also initiates an XA transaction by initiating the money transfer from the customer's bank to the cinema's bank.
- Customer's bank service: This service participates in the XA transaction and withdraws an amount from the customer's bank account.
- Cinema's bank service: This service participates in the XA transaction and withdraws an amount from the customer's bank account.

Example Sequence of Transaction Flow in a Nested Transaction

The following steps describe the successful path of LRA and XA transactions among the sample microservices. In case of failures, Payment service calls rollback instead of commit. If the payment fails, the Seat booking service removes the reservation on the cinema seat and makes it available for booking once more.

1. The Seat booking service initiates a new LRA transaction, when a user places a request to book a seat. The transaction initiator service, Seat booking, makes a call to MicroTx to start an LRA transaction.
2. After reserving the seat, the Seat booking service calls the Payment service to handle the payment.
3. Payment service, joins the existing LRA transaction, as a transaction participant.
4. Next, the Payment service acts as an XA initiator service to start an XA transaction by making a call to MicroTx. It initiates the transfer of money from the customer's bank service to the cinema's bank service.
5. Customer's bank service enlists to the XA transaction, performs a DML operation to withdraw the amount, and then returns a response.
6. The Payment service initiates another request to deposit an amount to Cinema's bank service.
7. The XA transaction participant service, Cinema's bank enlists to the XA transaction, performs a DML operation to deposit the amount, and then returns a response.
8. The Payment service commits the XA transaction only if the both the requests, that is, the request to Customer's bank and the request to Cinema's bank, are executed successfully. In case of any failure, the Payment service calls rollback instead of commit.
9. After committing the XA transaction, the Payment service returns the payment status to the Seat booking service.
10. The Seat booking service calls the MicroTx to complete the LRA transaction if the payment is successful.
11. Transaction Manager for Microservices calls the complete callback URI of Payment service and Seat booking service to complete the LRA transaction and book the seat in cinema.
12. The Seat booking service returns details of the booked seat to the user.

7.2.2 Run Sample LRA Application in Kubernetes

- Build Docker Images for Sample LRA Application
  The LRA sample application is available in the installation bundle in the `installation_directory/otmm-<version>/samples/lra/lrademo` folder. This folder contains the code for three microservices, YAML file, and Helm charts.
- Push LRA Sample App Images
  Push the Docker image of the sample applications, that you have built, to a remote repository.
- Update the values.yaml File for LRA
  The sample application files also contain the `values.yaml` file, the manifest file of the sample application, which contains the deployment configuration details for the LRA sample application.
- Install LRA Sample Application
  Install the LRA sample application in the Kubernetes cluster where you have installed MicroTx.
- Run an LRA Transaction

7.2.2.1 Build Docker Images for Sample LRA Application

The LRA sample application is available in the installation bundle in the `installation_directory/otmm-<version>/samples/lra/lrademo` folder. This folder contains the code for three microservices, YAML file, and Helm charts.

For details about the sample LRA application, see About the Sample LRA Application.

Perform the following steps to build Docker images for each microservice in the sample:

1. Run the following commands to build the Docker image for the hotel application.

   ```
   cd installation_directory/otmm-<version>/samples/lra/lrademo/hotel
   docker image build -t hotel:1.0 .
   ```

   When the image is successfully built, the following message is displayed.

   Successfully tagged hotel:1.0

2. Run the following commands to build the Docker image for the flight application.

   ```
   cd installation_directory/otmm-<version>/samples/lra/lrademo/flight
   docker image build -t flight:1.0 .
   ```

   When the image is successfully built, the following message is displayed.

   Successfully tagged flight:1.0

3. Run the following commands to build the Docker image for the trip manager application.

   ```
   cd installation_directory/otmm-<version>/samples/lra/lrademo/trip-
   manager
   docker image build -t trip-manager:1.0 .
   ```

   When the image is successfully built, the following message is displayed.

   Successfully tagged trip-manager:1.0

The Docker images that you have created are available in your local Docker container registry.

7.2.2.2 Push LRA Sample App Images

Push the Docker image of the sample 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 the sample application.

1. In a terminal window on the client machine running Docker, log in to Oracle Cloud Infrastructure Registry, to which you want to push the image, by entering:

   ```
   docker login <region-key>.ocir.io
   ```

where `<region-key>` is the key for the Oracle Cloud Infrastructure Registry region you're using. For example, `phx`. See the Availability by Region topic in the Oracle Cloud Infrastructure Registry documentation.

2. 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.

   Sample commands

   The following sample commands tag the images of hotel, flight, and trip manager applications.

   ```
   docker tag hotel:1.0 <region-key>.ocir.io/otmmrepo/hotel:1.0
   docker tag trip-manager:1.0 <region-key>.ocir.io/otmmrepo/trip-
   manager:1.0
   docker tag flight:1.0 <region-key>.ocir.io/otmmrepo/flight:1.0
   ```

   Where, `<region-key>.ocir.io/otmmrepo` is the Oracle Cloud Infrastructure Registry to which you want to push the image file. If you are using other Kubernetes platforms, then provide the details based on your environment.

3. Push the Docker image from your local repository to the remote Docker repository.

   Syntax

   ```
   docker push remote_image[:tag]
   ```

   Sample commands

   The the following sample commands push the tagged images of hotel, flight, and trip manager applications.

   ```
   docker push <region-key>.ocir.io/otmmrepo/hotel:1.0
   docker push <region-key>.ocir.io/otmmrepo/trip-manager:1.0
   docker push <region-key>.ocir.io/otmmrepo/flight:1.0
   ```

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.

7.2.2.3 Update the values.yaml File for LRA

The sample application files also contain the `values.yaml` file, the manifest file of the sample application, which contains the deployment configuration details for the LRA sample application.

While deploying the sample application to a Kubernetes cluster, Helm pulls the sample application images 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 and environment details in the `values.yaml` file:

1. Open the `values.yaml` file, which is located at *`installation_directory`*`/otmm-`*`RELEASE`*`/samples/lra/helmcharts/sampleappslra/values.yaml`, in any code editor. This file contains sample values.
2. Provide details of all the sample application images that you have uploaded to the docker container. For example, `iad.ocir.io/mytenancy/lra/trip-manager-lra:v1`.
3. Save your changes.

7.2.2.4 Install LRA Sample Application

Install the LRA sample application in the Kubernetes cluster where you have installed MicroTx.

1. Run the following commands to install the LRA sample application.

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

   helm install sample-lra-app --namespace otmm sampleappslra/ \
   --values sampleappslra/values.yaml
   ```

   Where *sample-lra-app* is the name of the application that is installed.

   The following output is displayed.

   ```
   NAME: sample-lra-app
   LAST DEPLOYED: Wed Apr 20 17:12:32 2022
   NAMESPACE: otmm
   STATUS: deployed
   REVISION: 1
   TEST SUITE: None
   ```

2. 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
   ```

   The following sample output shows that all the pods are ready and in the `Running` state.

   ```
   NAME                                    READY   STATUS    RESTARTS
   AGE
   pod/flight-95db44488-h4br8              2/2     Running   0
   17h
   pod/hotel-75bd8c59cb-hxgj5              2/2     Running   0
   17h
   pod/otmm-tcs-84b87b66bd-9mntz           2/2     Running   1 (20h ago)
   37h
   pod/trip-manager-6df68db55b-sdhcg       2/2     Running   0
   17h
   ```

```
NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)    AGE
service/flight          ClusterIP   10.100........  <none>
8080/TCP   17h
service/hotel           ClusterIP   10.101........  <none>
8080/TCP   17h
service/otmm-tcs        ClusterIP   10.109........  <none>
9000/TCP   37h
service/trip-manager    ClusterIP   10.97.........  <none>
8080/TCP   17h

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/flight          1/1     1            1           17h
deployment.apps/hotel           1/1     1            1           17h
deployment.apps/otmm-tcs        1/1     1            1           37h
deployment.apps/trip-manager    1/1     1            1           17h

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/flight-95db44488           1         1         1       17h
replicaset.apps/hotel-75bd8c59cb           1         1         1       17h
replicaset.apps/otmm-tcs-84b87b66bd        1         1         1       37h
replicaset.apps/trip-manager-6df68db55b    1         1         1       17h
```

3. Verify that the application is installed.

   ```
   helm list --namespace otmm
   ```

   The following sample output displays details of the applications installed in the `otmm` namespace. Where, `sample-lra-app` is the LRA sample application that you have installed.

   ```
   NAME            NAMESPACE       REVISION
   UPDATED                                 STATUS
   CHART                   APP VERSION
   otmm            otmm            1               2022-04-19
   21:14:25.1941414 +0530 IST   deployed        otmm-RELEASE  RELEASE
   sample-lra-app  otmm            1               2022-04-20
   17:12:32.8553506 +0530 IST   deployed        sampleappslra-1.0.1     1.0.1
   ```

7.2.2.5 Run an LRA Transaction

Before you start a transaction, you must create an access token, install the MicroTx library files, and note down the external IP address of the Istio ingress gateway.

1. Run the following command to book a hotel and flight.

   Command Syntax

   ```
   curl
   -H "Authorization:Bearer $TOKEN"
   -X POST
   -d '' external-IP-address-Istio-ingress-gateway:Istio port number/
   application-specific-URI-for-transaction
   ```

Sample Command

```
curl
-H "Authorization:Bearer $TOKEN"
-X POST
-d '' "https://192.0.2.1:443/trip-service/api/trip?
hotelName=Mercury&flightNumber=A123" | jq
```

Where,

- `192.0.2.1` is the external IP address of the Istio ingress gateway.
- `443` is the Istio port number.
- *`TOKEN`* is the name of the variable in which you stored the authentication token earlier. For information about retrieving the authentication token and storing it in a variable, see Create an Access Token.

Sample Response

```
{
  "cancelPending": false,
  "details": [
    {
      "cancelPending": false,
      "details": [

      ],
      "encodedId": "http%3A%2F%2Fomtm-tcs%3A9000%2Fapi%2Fv1%2Flra-
coordinator%2F011899ca-20f3-4d8c-9e92-76de355921fe",
      "id": "http://otmm-tcs:9000/api/v1/lra-coordinator/
011899ca-20f3-4d8c-9e92-76de355921fe",
      "name": "Mercury",
      "status": "PROVISIONAL",
      "type": "Hotel"
    },
    {
      "cancelPending": false,
      "details": [

      ],
     "encodedId": "http%3A%2F%2Fomtm-tcs%3A9000%2Fapi%2Fv1%2Flra-
coordinator%2F011899ca-20f3-4d8c-9e92-76de355921fe",
      "id": "http://otmm-tcs:9000/api/v1/lra-coordinator/
011899ca-20f3-4d8c-9e92-76de355921fe",
      "name": "A123",
      "status": "PROVISIONAL",
      "type": "Flight"
    }
  ],
  "encodedId": "http%3A%2F%2Fomtm-tcs%3A9000%2Fapi%2Fv1%2Flra-
coordinator%2F011899ca-20f3-4d8c-9e92-76de355921fe",
  "id": "http://otmm-tcs:9000/api/v1/lra-coordinator/
011899ca-20f3-4d8c-9e92-76de355921fe",
  "name": "Aggregate Booking",
  "status": "PROVISIONAL",
  "type": "Trip"
}
```

2. Note down the value of the `encodedId` and `id`. You will need to provide this information.
3. Run the following command to confirm the transaction.

   Command Syntax

   ```
   curl --location
   -H "Authorization:Bearer $TOKEN"
   -H "Long-Running-Action: LRA-ID"
   --request PUT
   -d '' http://external-ip-Istio-ingress-gateway/trip-service/api/trip/url-
   encoded-LRA-ID
   ```

   Where, `LRA-ID` is the value of the `id` attribute and `url-encoded-LRA-ID` is the value of the `encodedId` attribute that you have noted down.

   Sample Command

   ```
   curl --location
   -H "Authorization:Bearer $TOKEN"
   -H "Long-Running-Action: http://otmm-tcs:9000/lra-coordinator/
   011899ca-20f3-4d8c-9e92-76de355922fe"
   --request PUT
   -d '' "https://192.0.2.1:443/trip-service/api/trip/http%3A%2F%2Fotmm-
   tcs%3A9000%2Flra-coordinator%2F011899ca-20f3-4d8c-9e92-76de355921fe"
   ```

4. Run the following commands to see the status of the booking.

   ```
   curl -X GET -H "Authorization:Bearer $TOKEN" https://192.0.2.1:443/
   hotelService/api/hotel | jq
   curl -X GET -H "Authorization:Bearer $TOKEN" https://192.0.2.1:443/
   flightService/api/flight | jq
   ```

7.2.3 Run Sample LRA Application in Docker Swarm

- Build and Push the Docker Images
  The LRA sample application is available in the installation bundle in the `installation_directory/otmm-<version>/samples/lra/lrademo` folder. This folder contains the application code for three sample microservices which are used to book a hotel and flight ticket.
- Install LRA Sample Application
  Install the LRA sample application in Docker Swarm.
- Run an LRA Transaction

7.2.3.1 Build and Push the Docker Images

The LRA sample application is available in the installation bundle in the `installation_directory/otmm-<version>/samples/lra/lrademo` folder. This folder contains the application code for three sample microservices which are used to book a hotel and flight ticket.

For details about the sample LRA application, see About the Sample LRA Application. It is important that you tag the Docker images that you build with the address of the registry that you have created. For example, *198.51.100.1:5000*. This is required while distributing the apps to the swarm.

Perform the following steps to build Docker images for each microservice in the sample:

1. Run the following commands to build the Docker image for the hotel application.

   ```
   cd installation_directory/otmm-<version>/samples/lra/lrademo/hotel
   docker image build -t 198.51.100.1:5000/hotel:1.0 .
   ```

   Where, *198.51.100.1:5000* is the address of the Docker registry that you have created.

2. Run the following commands to build the Docker image for the flight application.

   ```
   cd installation_directory/otmm-<version>/samples/lra/lrademo/flight
   docker image build -t 198.51.100.1:5000/flight:1.0 .
   ```

3. Run the following commands to build the Docker image for the trip manager application.

   ```
   cd installation_directory/otmm-<version>/samples/lra/lrademo/trip-
   manager
   docker image build -t 198.51.100.1:5000/trip-manager:1.0 .
   ```

4. Push the tagged Docker image to the Docker registry that you have created.

   Syntax

   ```
   docker push image[:tag]
   ```

   Sample commands

   The the following sample commands push the tagged images of hotel, flight, and trip manager applications.

   ```
   docker push 198.51.100.1:5000/hotel:1.0
   docker push 198.51.100.1:5000/trip-manager:1.0
   docker push 198.51.100.1:5000/flight:1.0
   ```

   When you build the Docker images, they are available in your local Docker container registry. When you push the Docker image, it becomes available in the docker registry that you have created for the swarm.

5. Ensure that Java Development Kit (JDK) is installed on your local system, and then run the following commands in the Bash shell to set the following environment variables.

   ```
   export JAVA_HOME=jdk-install-dir
   export PATH=$JAVA_HOME/bin:$PATH
   ```

6. Build the Trip client application which you can use to send a request to book a new trip.

   ```
   cd installation_directory/otmm-<version>/samples/lra/lrademo/trip-client
   mvn clean package
   ```

7.2.3.2 Install LRA Sample Application

Install the LRA sample application in Docker Swarm.

All Swarm objects are described in manifests called *stack files*. The `tmm-stack-compose.yaml` stack file is located at `installation_directory/otmm-<version>/samples/docker`. This YAML file describes all the components and configurations of the LRA sample application and transaction coordinator. Use this file to run and manage the microservices in Docker Swarm.

To run LRA sample application:

1. Deploy the `tmm-stack-compose.yaml` stack file.

   ```
   cd installation_directory/otmm-<version>/samples/docker
   docker stack deploy -c tmm-stack-compose.yaml tmmdemo
   ```

   Where, `tmmdemo` is the name of the Docker stack that you want to install. You can specify any other name.

   ```
   Output:
   Creating network tmmdemo_default
   Creating config tmmdemo_my_tcs_config
   Creating service tmmdemo_hotel
   Creating service tmmdemo_flight
   Creating service tmmdemo_trip-manager
   Creating service tmmdemo_otmm-tcs
   ```

2. Verify that all services are ready. Use the following command to retrieve the list of services and their status.

   ```
   docker service ls
   ```

   The following sample output shows that all the services are ready.

   ```
   ID             NAME                   MODE         REPLICAS
   IMAGE                             PORTS
   tjc0u55yavu4   registry               replicated   1/1
   registry:2                        *:5000->5000/tcp
   qvzeovz8729y   tmmdemo_flight         replicated   1/1
   198.51.100.1:5000/flight:1.0         *:8083->8083/tcp
   ifmqd521im28   tmmdemo_hotel          replicated   1/1
   198.51.100.1:5000/hotel:1.0          *:8082->8082/tcp
   ilkvx4emyv8c   tmmdemo_otmm-tcs       replicated   1/1
   198.51.100.1:5000/tmm:latest         *:9000->9000/tcp
   m069vayql490   tmmdemo_trip-manager   replicated   1/1
   198.51.100.1:5000/trip-manager:1.0   *:8081->8081/tcp
   ```

Note down the port numbers on which the applications are running as you will need to provide the port number when you run the sample application.

When the services are ready, you can run an LRA transaction.

7.2.3.3 Run an LRA Transaction

To run the sample LRA application to book a hotel room and flight ticket.

1. Set the URL for the Trip Manager service.

   Syntax

   ```
   export TRIP_SERVICE_URL=<IP-address-of-Docker-registry>:<port-of-
   sample-app>/trip-service/api/trip
   ```

   Example

   ```
   export TRIP_SERVICE_URL=http://198.51.100.1:8081/trip-service/api/
   trip
   ```

   Where,

   - `198.51.100.1` is the IP address of the Docker registry to which you have pushed the Docker images.
   - `8081` is the port number on which the Trip Manager service is running.

   Provide these details based on your environment.

2. Run the Trip Client application.

   ```
   cd installation_directory/otmm-<version>/samples/lra/lrademo/trip-
   client
   java -jar target/trip-client.jar
   ```

   The Trip Booking Service console is displayed.

3. Type **y** to confirm that you want to run the LRA sample application, and then press Enter.

   The sample application provisionally books a hotel room and a flight ticket and displays the details of the provisional booking.

4. Type **y** to confirm the provisional booking, and then press Enter.

   Your booking is confirmed and information about your confirmed booking is displayed.

5. To retrieve the details of your booking, run the following command.

   ```
   curl --location --request GET http://198.51.100.1:8081/trip-
   service/api/trip | jq
   ```

   Where,

   - `198.51.100.1` is the IP address of the Docker registry to which you have pushed the Docker images.

- `8081` is the port number on which the Trip Booking service is running.

Sample Response

```
[
  {
    "details": [
      {
        "encodedId": "http%3A%2F%2Fotmm-tcs%3A9000%2Fapi%2Fv1%2Flra-
coordinator%2F9c44a549-9047-41d3-a3f0-623da46c6b2b",
        "id": "http://otmm-tcs:9000/api/v1/lra-coordinator/
9c44a549-9047-41d3-a3f0-623da46c6b2b",
        "name": "Acme",
        "status": "CONFIRMED",
        "type": "Hotel"
      },
      {
        "details": [],
        "encodedId": "http%3A%2F%2Fotmm-tcs%3A9000%2Fapi%2Fv1%2Flra-
coordinator%2F9c44a549-9047-41d3-a3f0-623da46c6b2b",
        "id": "http://otmm-tcs:9000/api/v1/lra-coordinator/
9c44a549-9047-41d3-a3f0-623da46c6b2b",
        "name": "A123",
        "status": "CONFIRMED",
        "type": "Flight"
      }
    ],
    "encodedId": "http%3A%2F%2Fotmm-tcs%3A9000%2Fapi%2Fv1%2Flra-
coordinator%2F9c44a549-9047-41d3-a3f0-623da46c6b2b",
    "id": "http://otmm-tcs:9000/api/v1/lra-coordinator/9c44a549-9047-41d3-
a3f0-623da46c6b2b",
    "name": "Trip",
    "status": "CONFIRMED",
    "type": "Trip"
  }
]
```

6. Run the following commands to see the list of hotel bookings.

Sample Command

```
curl --location --request GET http://198.51.100.1:8082/hotelService/api/
hotel | jq
```

Where,

- `198.51.100.1` is the IP address of the Docker registry to which you have pushed the Docker images.
- `8082` is the port number on which the Hotel Booking service is running.

Provide these details based on your environment.

Sample Response

```
[
  {
    "encodedId": "http%3A%2F%2Fotmm-tcs%3A9000%2Fapi%2Fv1%2Flra-
coordinator%2F9c44a549-9047-41d3-a3f0-623da46c6b2b",
    "id": "http://otmm-tcs:9000/api/v1/lra-coordinator/
9c44a549-9047-41d3-a3f0-623da46c6b2b",
    "name": "Acme",
    "status": "CONFIRMED",
    "type": "Hotel"
  }
]
```

Note down the encoded ID. You will need to provide this value if you want to retrieve details of a specific flight or hotel booking.

7. Run the following commands to see the list of flight bookings.

   Sample Commands

   ```
   curl --location --request GET http://198.51.100.1:8083/
   flightService/api/flight | jq
   ```

   Where,

   - `198.51.100.1` is the IP address of the Docker registry to which you have pushed the Docker images.
   - `8083` is the port number on which the Flight Booking service is running respectively.

   Provide these details based on your environment.

   Sample Response

   ```
   [
     {
       "details": [],
       "encodedId": "http%3A%2F%2Fotmm-tcs%3A9000%2Fapi%2Fv1%2Flra-
   coordinator%2F9c44a549-9047-41d3-a3f0-623da46c6b2b",
       "id": "http://otmm-tcs:9000/api/v1/lra-coordinator/
   9c44a549-9047-41d3-a3f0-623da46c6b2b",
       "name": "A123",
       "status": "CONFIRMED",
       "type": "Flight"
     }
   ]
   ```

   Note down the encoded ID. You will need to provide this value if you want to retrieve details of a specific flight or hotel booking.

8. Run the following commands to see the details of a specific trip, hotel, or flight booking. You can specify the encoded ID of the booking for which you want to retrieve the details.

   Command Syntax

   ```
   curl --location --request GET http://198.51.100.1:8081/trip-
   service/api/trip/<encodedId> | jq
   curl --location --request GET http://198.51.100.1:8082/
   ```

```
hotelService/api/hotel/<encodedId> | jq
curl --location --request GET http://198.51.100.1:8083/flightService/api/
flight/<encodedId> | jq
```

Sample Command

The following command retrieves the trip details of the specified encoded ID.

```
curl --location --request GET http://198.51.100.1:8081/trip-service/api/
trip/http%3A%2F%2Fotmm-tcs%3A9000%2Fapi%2Fv1%2Flra-
coordinator%2F9c44a549-9047-41d3-a3f0-623da46c6b2b | jq
```

7.3 Deploy TCC Sample Application

- 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.
- Run Sample TCC Application in Kubernetes
- Run Sample TCC Application in Docker Swarm

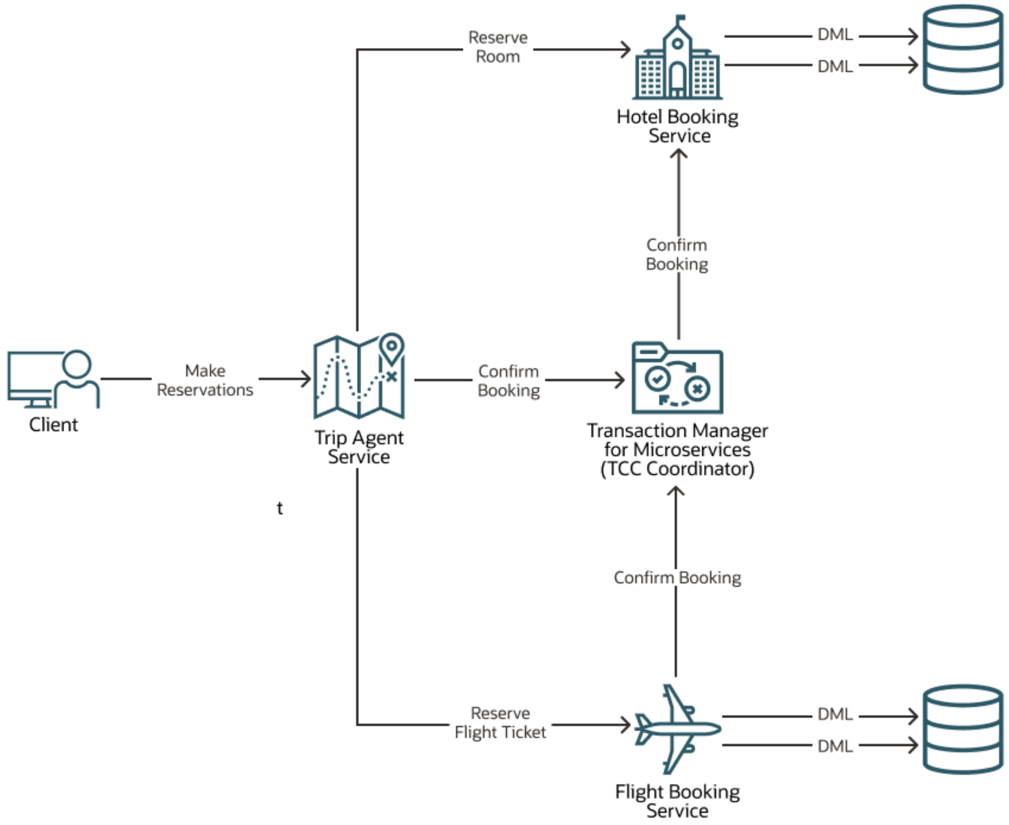
7.3.1 About the Sample TCC Application

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 TCC sample application files are available in the installation bundle in the `installation_directory/otmm-<version>/samples/tcc` folder. This folder contains the code for three microservices, YAML file, and Helm charts.
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. The Java application is located at `installation_directory/otmm-<version>/samples/tcc/java/travel-agent` and the Node.js application is located at `installation_directory/otmm-<version>/samples/tcc/nodejs/travel-agent`.
- 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. The Java application is located at `installation_directory/otmm-<version>/samples/tcc/java/hotel-booking` and the Node.js application is located at `installation_directory/otmm-<version>/samples/tcc/nodejs/hotel-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 Java application is located at `installation_directory/otmm-<version>/samples/tcc/java/flight-booking` and the Node.js application is located at `installation_directory/otmm-<version>/samples/tcc/nodejs/flight-booking`.

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.

7.3.2 Run Sample TCC Application in Kubernetes

- Build Docker Images for Sample TCC Application
  The TCC sample application is available in the installation bundle in the `installation_directory/otmm-RELEASE/samples/tcc` folder.
- Push TCC Sample App Images
  Push the Docker image of the sample applications, that you have built, to a remote repository.
- Update the values.yaml File for TCC
  The sample application folder also contain the `values.yaml` file, the manifest file of the sample application, which contains the deployment configuration details for the TCC sample application.
- Install TCC Sample Application
  Install the TCC sample application in the Kubernetes cluster where you have installed MicroTx.
- Run a TCC Transaction
  When you run the application, it makes a provisional booking by reserving a hotel room and flight ticket.

7.3.2.1 Build Docker Images for Sample TCC Application

The TCC sample application is available in the installation bundle in the `installation_directory/otmm-RELEASE/samples/tcc` folder.

This folder contains individual folders for the sample code written in the Java, Node.js, and Python language. The folder for sample application in each language contains code files for the three microservices, YAML file, and Helm charts. Decide which sample application you would like to run, and then build the Docker images for the language that you have chosen. For details about the sample TCC application, see About the Sample TCC Application.

Perform only one of the following steps to build the sample code to create Docker image for each microservice in the sample.

- Run the following commands to build the Docker images for the Java sample application.
  - Run the following command to build the flight application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/java/flight-booking
    docker image build -t flight-booking:1.0 .
    ```

  - Run the following command to build the hotel application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/java/hotel-booking
    docker image build -t hotel-booking:1.0 .
    ```

  - Run the following command to build the travel agent application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/java/travel-agent
    docker image build -t travel-agent:1.0 .
    ```

- Run the following commands to build the Docker images for the Python sample application.
  - Run the following command to build the flight application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/python/flight-
    booking-py
    docker image build -t flight-booking-py:1.0 .
    ```

  - Run the following command to build the hotel application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/python/
    hotel_booking-py
    docker image build -t hotel-booking-py:1.0 .
    ```

  - Run the following command to build the travel agent application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/python/travel-
    agent-py
    docker image build -t travel-agent-py:1.0 .
    ```

- Run the following commands to build the Docker images for the Node.js sample application.
  - Run the following command to build the flight application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/nodejs/flight
    docker image build -t flight:1.0 .
    ```

  - Run the following command to build the hotel application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/nodejs/hotel
    docker image build -t hotel:1.0 .
    ```

  - Run the following command to build the travel agent application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/nodejs/travel-
    agent
    docker image build -t travel-agent:1.0 .
    ```

The Docker images that you have created are available in your local Docker container registry. Note down the names of the Docker images that you have created as you will have to provide these names in the next step.

7.3.2.2 Push TCC Sample App Images

Push the Docker image of the sample 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 the sample application.

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. 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. Provide the name of the Docker images in your local repository which you have noted down in the previous task.
   - *remote_image[:tag]* is the tag with which you want to identify the image in the remote Docker repository.

   Based on the language of the sample language, run one of the following commands.

   - The following sample commands tag the images of the hotel, flight, and trip manager Java applications.
     Sample commands

     ```
     docker tag hotel-booking:1.0 <region-key>.ocir.io/otmmrepo/hotel-
     booking:1.0
     docker tag flight-booking:1.0 <region-key>.ocir.io/otmmrepo/flight-
     booking:1.0
     docker tag travel-agent:1.0 <region-key>.ocir.io/otmmrepo/travel-
     agent:1.0
     ```

   - The following sample commands tag the images of the hotel, flight, and trip manager Python applications.
     Sample commands

     ```
     docker tag hotel_booking-py:1.0 <region-key>.ocir.io/otmmrepo/
     hotel_booking-py:1.0
     docker tag flight_booking-py:1.0 <region-key>.ocir.io/otmmrepo/
     flight_booking-py:1.0
     docker tag travel-agent-py:1.0 <region-key>.ocir.io/otmmrepo/travel-
     agent-py:1.0
     ```

   - The following sample commands tag the images of the hotel, flight, and trip manager Node.js applications.
     Sample commands

     ```
     docker tag hotel:1.0 <region-key>.ocir.io/otmmrepo/hotel:1.0
     docker tag flight:1.0 <region-key>.ocir.io/otmmrepo/flight:1.0
     docker tag travel:1.0 <region-key>.ocir.io/otmmrepo/travel-agent:1.0
     ```

   Where, `<region-key>.ocir.io/otmmrepo` is the Oracle Cloud Infrastructure Registry to which you want to push the image file. If you are using other Kubernetes platforms, then provide the details based on your environment.

3. Push the Docker image from your local repository to the remote Docker repository.

   Syntax

   ```
   docker push remote_image[:tag]
   ```

Sample commands

The the following sample commands push the tagged images of hotel, flight, and trip manager applications. Provide the names of the remote images based on the information that you have entered in the previous step.

```
docker push <region-key>.ocir.io/otmmrepo/hotel-booking:1.0
docker push <region-key>.ocir.io/otmmrepo/travel-agent:1.0
docker push <region-key>.ocir.io/otmmrepo/flight-booking:1.0
```

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.

7.3.2.3 Update the values.yaml File for TCC

The sample application folder also contain the `values.yaml` file, the manifest file of the sample application, which contains the deployment configuration details for the TCC sample application.

When you use Helm to deploy the sample application to a Kubernetes cluster, Helm pulls the sample application images from the remote Docker registry based on the details provided in the `values.yaml` file. Update the `values.yaml` file to specify the names of the Docker images.

To update the names of the Docker images that you have pushed to the remote repository in the `values.yaml` file:

1. Open the `values.yaml` file in any code editor. This file contains sample values.

   For the Java sample application, the file is located at *installation_directory*/otmm-*RELEASE*/samples/tcc/java/helmcharts/sampleappstcc/values.yaml.

   For the Node.js sample application, the file is located at *installation_directory*/otmm-*RELEASE*/samples/tcc/nodejs/helmcharts/sampleappstccnode/values.yaml.

   For the Python sample application, the file is located at *installation_directory*/otmm-*RELEASE*/samples/tcc/python/helmcharts/sampleappstccpy/values.yaml.
2. Provide details of all the sample application images that you have uploaded to the remote Docker repository. For example, `iad.ocir.io/mytenancy/tcc/flight-booking-tcc:v1`.
3. Save your changes.

7.3.2.4 Install TCC Sample Application

Install the TCC sample application in the Kubernetes cluster where you have installed MicroTx.

1. Install the TCC sample application.
   - Run the following commands to install the Java sample application.

     ```
     cd installation_directory/otmm-RELEASE/otmm/samples/tcc/java/
     helmcharts
     ```

```
helm install sample-tcc-app --namespace otmm sampleappstcc/ \
--values sampleappstcc/values.yaml
```

- Run the following commands to install the Node.js sample application.

  ```
  cd installation_directory/otmm-RELEASE/otmm/samples/tcc/nodejs/
  helmcharts
  ```

  ```
  helm install sample-tcc-app --namespace otmm sampleappstccnode/ \
  --values sampleappstccnode/values.yaml
  ```

- Run the following commands to install the Python sample application.

  ```
  cd installation_directory/otmm-RELEASE/otmm/samples/tcc/python/
  helmcharts
  ```

  ```
  helm install sample-tcc-app --namespace otmm sampleappstccpy/ \
  --values sampleappstccpy/values.yaml
  ```

Where *sample-tcc-app* is the name of the application that is installed.

2. 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
   ```

3. Verify that the application is installed.

   ```
   helm list --namespace otmm
   ```

7.3.2.5 Run a TCC Transaction

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 provisional booking of the hotel room and flight ticket is canceled. In case of a cancellation, your application must include the code for releasing the provisionally blocked hotel and flight and making these resources available.

Before you start a transaction, you must create an access token, install the MicroTx library files, and note down the external IP address of the Istio ingress gateway.

To run the TCC sample application:

1. Run the following command to reserve a hotel and flight booking.

   Sample Command

   ```
   curl -H "Authorization:Bearer $TOKEN" \
        --header 'Accept: application/json' \
        -X POST \
        -d '' "https://$CLUSTER_IPADDR/travel-agent/api/bookings/reserve?
   hotelName=Acme&flightNumber=AA2250"
   ```

Where,

- *CLUSTER_IPADDR* is the name of the variable in which you stored the external IP address of the Istio ingress gateway. For information about finding the external IP address of the Istio ingress gateway and storing it in a variable, see Find IP Address of Istio Ingress Gateway.
- *TOKEN* is the name of the variable in which you stored the authentication token earlier. For information about retrieving the authentication token and storing it in a variable, see Create an Access Token.

Sample Response

```
{
    "tripBookingId": "840c7f0c-d87e-4694-aba5-0846e716ce99",
    "message": "Successfully booked the trip",
    "status": "RESERVED",
    "flightBooking": {
        "bookingId": "e32e1cbf-4d6d-431a-a5af-d48570e02666",
        "bookingUri": "http://$CLUSTER_IPADDR/travel-agent/api/
bookings/e32e1cbf-4d6d-431a-a5af-d48570e02666",
        "expires": 120000,
        "name": "AA2250",
        "startTime": 1677146471233,
        "type": "FLIGHT"
    },
    "hotelBooking": {
        "bookingId": "e140cdba-30a6-44c0-b7c2-c168f763641c",
        "bookingUri": "http://$CLUSTER_IPADDR/travel-agent/api/
bookings/e140cdba-30a6-44c0-b7c2-c168f763641c",
        "expires": 120000,
        "name": "Acme",
        "startTime": 1677146471209,
        "type": "HOTEL"
    }
}
```

This commands reserves a hotel and flight booking and the status is `RESERVED`.

2. Note down the values of `tripBookingId` and the `link` response header. You will need to provide this information in the next step.
3. You can choose to either confirm or cancel the reservation. Run one of the following commands to confirm or cancel the transaction.
   - To confirm a transaction, run the following command:
     Command Syntax

     ```
     curl --location --request PUT -H "Authorization:Bearer $TOKEN" \
          -d '' http://$CLUSTER_IPADDR/travel-agent/api/confirm/
     tripBookingId
     ```

     Sample Command

     ```
     curl -H "Authorization:Bearer $TOKEN" \
          --header 'Accept: application/json' \
     ```

```
     -H "link: <http://192.0.4.1:9000/api/v1/tcc-transaction/4e6dc225-
d8af-4988-8446-a70e4cbd1e44>; rel=\"https://otmm.oracle.com/tcc-
transaction\""
     -X PUT \
     -d '' "https://$CLUSTER_IPADDR/travel-agent/api/confirm/840c7f0c-
d87e-4694-aba5-0846e716ce99"
```

- To cancel a transaction, run the following command:
  Command Syntax

  ```
  curl --location --request PUT -H "Authorization:Bearer $TOKEN" \
       -d '' https://external-IP-address-Istio-ingress-gateway/travel-
  agent/api/cancel/tripBookingId
  ```

  Sample Command

  ```
  curl -H "Authorization:Bearer $TOKEN" \
       --location \
       --header 'Accept: application/json' \
       -H "link: <http://192.0.4.1:9000/api/v1/tcc-transaction/4e6dc225-
  d8af-4988-8446-a70e4cbd1e44>; rel=\"https://otmm.oracle.com/tcc-
  transaction\""
       -X DELETE \
       -d '' "https://$CLUSTER_IPADDR/travel-agent/api/cancel/840c7f0c-
  d87e-4694-aba5-0846e716ce99"
  ```

4. View the status and details of a single booking by provide its `tripBookingId`.

   Command Syntax

   ```
   curl --location --request GET -H "Authorization:Bearer $TOKEN" \
        https://external-IP-address-Istio-ingress-gateway/travel-agent/api/
   bookings/tripBookingId
   ```

   Sample Command

   ```
   curl -H "Authorization:Bearer $TOKEN" \
        --location \
        --header 'Accept: application/json' \
        -X GET \
        "https://$CLUSTER_IPADDR/travel-agent/api/bookings/840c7f0c-
   d87e-4694-aba5-0846e716ce99"
   ```

5. Run the following command to view the status and details of all bookings.

   Command Syntax

   ```
   curl --location --request GET -H "Authorization:Bearer $TOKEN" \
        https://external-IP-address-Istio-ingress-gateway/travel-agent/api/
   bookings
   ```

Sample Command

```
curl -H "Authorization:Bearer $TOKEN" \
     --location \
     --header 'Accept: application/json' \
     -X GET \
     "https://$CLUSTER_IPADDR/travel-agent/api/bookings"
```

7.3.3 Run Sample TCC Application in Docker Swarm

- Build Docker Images for Sample TCC Application
  The TCC sample application is available in the installation bundle in the `installation_directory/otmm-RELEASE/samples/tcc` folder.
- Install TCC Sample Application
  Install the TCC Sample Application in Docker Swarm.
- Run the Sample TCC Application
  When you run the application, it makes a provisional booking by reserving a hotel room and flight ticket.

7.3.3.1 Build Docker Images for Sample TCC Application

The TCC sample application is available in the installation bundle in the `installation_directory/otmm-RELEASE/samples/tcc` folder.

This folder contains individual folders for the sample code written in the Java, Node.js, and Python languages. The folder for sample application in each language contains code files for the three microservices, YAML file, and Helm charts. Decide which sample application you would like to run, and then build the Docker images for the language that you have chosen. For details about the sample TCC application, see About the Sample TCC Application.

1. Store the location of the Docker registry in an environment variable named `REGISTRY_LOCATION` as shown in the following command.

   ```
   export REGISTRY_LOCATION=192.0.2.1:5000
   ```

   Where,

   - `192.0.2.1` is the IP address of the Docker registry that you have created.
   - `5000` is the port number over which the Docker registry container communicates. Ensure that you have set up the required networking rules to permit inbound and outbound HTTPS or HTTP traffic over this port.

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

2. Based on whether you want to try out the Python, Java, or Node.js sample app, perform only one of the following steps to build the sample code to create Docker image for each microservice in the sample.

   - Run the following commands to build the Docker images for the Java sample application.

  - Run the following command to build the flight application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/java/flight-
    booking
    docker image build -t $REGISTRY_LOCATION/flight-booking:1.0 .
    ```

  - Run the following command to build the hotel application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/java/hotel-
    booking
    docker image build -t $REGISTRY_LOCATION/hotel-booking:1.0 .
    ```

  - Run the following command to build the travel agent application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/java/travel-agent
    docker image build -t $REGISTRY_LOCATION/travel-agent:1.0 .
    ```

- Run the following commands to build the Docker images for the Python sample application.
  - Run the following command to build the flight application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/python/flight-
    booking-py
    docker image build -t $REGISTRY_LOCATION/flight-booking-py:1.0 .
    ```

  - Run the following command to build the hotel application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/python/hotel-
    booking-py
    docker image build -t $REGISTRY_LOCATION/hotel-booking-py:1.0 .
    ```

  - Run the following command to build the travel agent application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/python/travel-
    agent-py
    docker image build -t $REGISTRY_LOCATION/travel-agent-py:1.0 .
    ```

- Run the following commands to build the Docker images for the Node.js sample application.
  - Run the following command to build the flight application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/nodejs/flight
    docker image build -t $REGISTRY_LOCATION/flight:1.0 .
    ```

  - Run the following command to build the hotel application.

    ```
    cd installation_directory/otmm-RELEASE/samples/tcc/nodejs/hotel
    docker image build -t $REGISTRY_LOCATION/hotel:1.0 .
    ```

- Run the following command to build the travel agent application.

  ```
  cd installation_directory/otmm-RELEASE/samples/tcc/nodejs/
  travel-agent
  docker image build -t $REGISTRY_LOCATION/travel-agent:1.0 .
  ```

3. Push the tagged Docker image to the Docker registry that you have created. Run one of the following commands based on the language of the sample application that you want to try out.

   When you build the Docker images, they are available in your local Docker container registry. When you push the Docker image, it becomes available in the docker registry that you have created for the swarm.

   Syntax

   ```
   docker push image[:tag]
   ```

   - The following sample commands tag the images of the hotel, flight, and trip manager Java applications.
     Sample commands

     ```
     docker push $REGISTRY_LOCATION/hotel-booking:1.0
     docker push $REGISTRY_LOCATION/flight-booking:1.0
     docker push $REGISTRY_LOCATION/travel-agent:1.0
     ```

   - The following sample commands tag the images of the hotel, flight, and trip manager Python applications.
     Sample commands

     ```
     docker push $REGISTRY_LOCATION/hotel-booking-py:1.0
     docker push $REGISTRY_LOCATION/flight-booking-py:1.0
     docker push $REGISTRY_LOCATION/travel-agent-py:1.0
     ```

   - The following sample commands tag the images of the hotel, flight, and trip manager Node.js applications.
     Sample commands

     ```
     docker push $REGISTRY_LOCATION/hotel:1.0
     docker push $REGISTRY_LOCATION/flight:1.0
     docker push $REGISTRY_LOCATION/travel-agent:1.0
     ```

Note down the names of the Docker images that you have created as you will have to update the names of the images in the YAML file in the next step.

7.3.3.2 Install TCC Sample Application

Install the TCC Sample Application in Docker Swarm.

All Swarm objects are described in manifests called *stack files*. The `tmm-stack-compose.yaml` stack file is located at `installation_directory/otmm-<version>/samples/docker`. This YAML file describes all the components and configurations of the TCC sample application and transaction coordinator. Use this file to run and manage the microservices in Docker Swarm.

To install the TCC sample application:

1. Provide details of all the sample application images that you have uploaded to the remote Docker repository. For example, `$REGISTRY_LOCATION/travel-agent:1.0`.
2. Save your changes.
3. Deploy the `tmm-stack-compose.yaml` stack file.

   ```
   cd installation_directory/otmm-<version>/samples/docker
   docker stack deploy -c tmm-stack-compose.yaml tmmtccdemo
   ```

   Where, `tmmtccdemo` is the name of the Docker stack that you want to install. You can specify any other name.
4. Verify that all services are ready. Use the following command to retrieve the list of services and their status.

   ```
   docker service ls
   ```

When the services are ready, you can run a TCC transaction.

7.3.3.3 Run the Sample TCC Application

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 provisional booking of the hotel room and flight ticket is canceled. Your application must include the code for releasing the provisionally blocked hotel and flight and making these resources available in case of a cancellation.

1. Run the following command to reserve a hotel and flight booking.

   Sample Command

   ```
   curl -H "Authorization:Bearer $TOKEN" \
        --header 'Accept: application/json' \
        -X POST \
        -d '' "https://$REGISTRY_LOCATION/travel-agent/api/bookings/reserve?
   hotelName=Acme&flightNumber=AA2250"
   ```

   Where,

   - *REGISTRY_LOCATION* is the name of the variable in which you stored the location of the Docker registry.
   - *TOKEN* is the name of the variable in which you stored the authentication token earlier. For information about retrieving the authentication token and storing it in a variable, see Create an Access Token.

   Sample Response

   ```
   {
       "tripBookingId": "840c7f0c-d87e-4694-aba5-0846e716ce99",
       "message": "Successfully booked the trip",
       "status": "RESERVED",
       "flightBooking": {
   ```

```
        "bookingId": "e32e1cbf-4d6d-431a-a5af-d48570e02666",
        "bookingUri": "http://$REGISTRY_LOCATION/travel-agent/api/
bookings/e32e1cbf-4d6d-431a-a5af-d48570e02666",
        "expires": 120000,
        "name": "AA2250",
        "startTime": 1677146471233,
        "type": "FLIGHT"
    },
    "hotelBooking": {
        "bookingId": "e140cdba-30a6-44c0-b7c2-c168f763641c",
        "bookingUri": "http://$REGISTRY_LOCATION/travel-agent/api/
bookings/e140cdba-30a6-44c0-b7c2-c168f763641c",
        "expires": 120000,
        "name": "Acme",
        "startTime": 1677146471209,
        "type": "HOTEL"
    }
}
```

This commands reserves a hotel and flight booking and the status is `RESERVED`.

2. Note down the values of `tripBookingId` and the `link` response header. You will need to provide this information in the next step.
3. You can choose to either confirm or cancel the reservation. Run one of the following commands to confirm or cancel the transaction.
   - To confirm a transaction, run the following command:
     Command Syntax

     ```
     curl --location --request PUT -H "Authorization:Bearer $TOKEN" \
          -d '' http://$REGISTRY_LOCATION/travel-agent/api/confirm/
     tripBookingId
     ```

     Sample Command

     ```
     curl -H "Authorization:Bearer $TOKEN" \
          --header 'Accept: application/json' \
          -H "link: <http://192.0.4.1:9000/api/v1/tcc-transaction/
     4e6dc225-d8af-4988-8446-a70e4cbd1e44>; rel=\"https://
     otmm.oracle.com/tcc-transaction\""
          -X PUT \
          -d '' "https://$REGISTRY_LOCATION/travel-agent/api/confirm/
     840c7f0c-d87e-4694-aba5-0846e716ce99"
     ```

   - To cancel a transaction, run the following command:
     Command Syntax

     ```
     curl --location --request PUT -H "Authorization:Bearer $TOKEN" \
          -d '' https://REGISTRY_LOCATION/travel-agent/api/cancel/
     tripBookingId
     ```

Sample Command

```
curl -H "Authorization:Bearer $TOKEN" \
     --location \
     --header 'Accept: application/json' \
     -H "link: <http://192.0.4.1:9000/api/v1/tcc-transaction/4e6dc225-
d8af-4988-8446-a70e4cbd1e44>; rel=\"https://otmm.oracle.com/tcc-
transaction\""
     -X DELETE \
     -d '' "https://$REGISTRY_LOCATION/travel-agent/api/cancel/
840c7f0c-d87e-4694-aba5-0846e716ce99"
```

4. View the status and details of a single booking by provide its `tripBookingId`.

   Command Syntax

   ```
   curl --location --request GET -H "Authorization:Bearer $TOKEN" \
        https://REGISTRY_LOCATION/travel-agent/api/bookings/tripBookingId
   ```

   Sample Command

   ```
   curl -H "Authorization:Bearer $TOKEN" \
        --location \
        --header 'Accept: application/json' \
        -X GET \
        "https://$REGISTRY_LOCATION/travel-agent/api/bookings/840c7f0c-
   d87e-4694-aba5-0846e716ce99"
   ```

5. Run the following command to view the status and details of all bookings.

   Command Syntax

   ```
   curl --location --request GET -H "Authorization:Bearer $TOKEN" \
        https://REGISTRY_LOCATION/travel-agent/api/bookings
   ```

   Sample Command

   ```
   curl -H "Authorization:Bearer $TOKEN" \
        --location \
        --header 'Accept: application/json' \
        -X GET \
        "https://$REGISTRY_LOCATION/travel-agent/api/bookings"
   ```

8 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, ORM, ORDS, Tuxedo, and WebLogic Server apps.

1. Before you begin, ensure that you have installed MicroTx and you can access it.
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 Java Apps with XA • Develop Node.js Apps with XA • Configure JPA or Hibernate App as Transaction Participant • Develop ORDS App as Transaction Participant • 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.
- Configure PostgreSQL as Resource Manager
  To use PostgreSQL as resource manager for XA transactions, you must enable prepared transactions and session affinity.
- 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.
- Subscribe to Receive XA Transaction Notifications
  From the MicroTx release 22.3.2, 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.
- Configure Library Properties
  Provide configuration information for the MicroTx library properties for every participant and initiator application.
- Develop Java Apps with XA
  Use the MicroTx library with your Java applications.
- Develop Node.js Apps with XA
- Develop ORDS App as Transaction Participant
  This section provides the detailed steps to configure a database application as an XA participant in the context of deploying and running the Oracle Database sample application.

8.1 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
  From the MicroTx release 22.3.1, you can optimize transactions where multiple transaction participant services use a single resource manager.

- Configure Multiple Resource Managers for a Single App
  From the MicroTx release 22.3.2, 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
  From MicroTx release 22.3.1, the transaction coordinator server resumes the transactions that were in progress when server the restarts after a failure.

8.1.1 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.

For Java XA transaction participant applications, the MicroTx library is tested with the following resource managers:

- Oracle Database 19c
- PostgreSQL 14.2
- MySQL and Microsoft SQL Server

For Node.js XA transaction participant applications, Transaction Manager for Microservices library is tested with Oracle Database v19.x.

XA transaction participant applications can use non-XA compliant resource managers, such as MongoDB 4.1 or later. For more information, see Optimizations for a Non-XA Resource.

8.1.2 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` 5.3.0 or above 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.

8.1.3 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.

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

From the MicroTx release 22.3.2, 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.

8.1.4 Common Resource Manager for Multiple Apps

From the MicroTx release 22.3.1, you can optimize transactions where multiple transaction participant services use a single resource manager.

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.

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.

8.1.5 Configure Multiple Resource Managers for a Single App

From the MicroTx release 22.3.2, 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.

8.1.6 About Dynamic Recovery for XA Transactions

From MicroTx release 22.3.1, 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, LRA, 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.

8.2 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. 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.

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

2. Restart the PostgreSQL service as shown in the following command.

   ```
   brew services restart postgresql
   ```

3. 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 for XA Participants.

8.3 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.

8.4 Subscribe to Receive XA Transaction Notifications

From the MicroTx release 22.3.2, 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.

8.5 Configure Library Properties

Provide configuration information for the MicroTx library properties for every participant and initiator application.

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 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`: 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`: From MicroTx release 22.3.1, you must 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.
- `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.
- 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.

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
```

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.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
```

Note that the environment variables names are case-sensitive.

8.6 Develop Java Apps with XA

Use the MicroTx library with your Java applications.

The MicroTx library for Java 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.
- Configure Java 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 Java 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 or Hibernate 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.

8.6.1 Configure Java 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 library as a maven dependency in the application's `pom.xml` file. The following sample code is for the 22.3 release. Provide the correct version, based on the release that you want to use.

```
<dependency>
     <groupId>com.oracle.tmm.jta</groupId>
     <artifactId>TmmLib</artifactId>
     <version>22.3</version>
</dependency>
```

3. Add the following code to the application code to include the `oracle.tmm.jta` package.

   ```
   package oracle.tmm.jta;
   ```

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.

   In the following code sample, you have created an instance `UserTransaction` of the `TrmUserTransaction` class. Define the methods for this object.

   ```
   public class TrmUserTransaction implements UserTransaction {
   // Define the methods for this object
   }
   ```

5. Add the following lines of code just before your application logic initiates or begins a transaction. The following code samples demonstrate how to create and begin a new XA transaction called `ut` by creating an instance of the `UserTransaction` object.

   - If your application only initiates the transaction and does not participate in the transaction, add the following lines to your application code.

     ```
     UserTransaction ut = new oracle.tmm.jta.TrmUserTransaction();
     ut.begin();
     ... // Implement the business logic to begin a transaction.
     ```

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

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

The sample XA application code for transaction initiator service is located at `*installation_directory*\otmm-*RELEASE*\samples\xa\java\accounts`. 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.
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 Java App as Transaction Participant .

8.6.2 Configure Java 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 Java App with an XA-Compliant Resource Manager
- Configure Java App with a Non-XA JDBC Resource
- Configure Java App with a Non-XA and Non-JDBC Resource

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

8.6.2.1 Configure Java App with an XA-Compliant Resource Manager

Use the information provided in this section to configure your Java 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 library as a maven dependency in the application's `pom.xml` file. The following sample code is for the 22.3 release. Provide the correct version, based on the release that you want to use.

```
<dependency>
     <groupId>com.oracle.tmm.jta</groupId>
     <artifactId>TmmLib</artifactId>
     <version>22.3</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 a participant microservice based on Spring Boot, register the `XAResourceCallbacks` (prepare/commit/rollback) and various filters as:

   ```
   @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.

9. Save the changes.

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

8.6.2.2 Configure Java App with Multiple XA-Compliant Resource Managers

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

Your application can connect to multiple XA-compliant resource managers. However, only a single non-XA resource can be a part of the transaction. If you are using multiple XA-compliant resource managers for your application, create a `.java` file to define property values and a `DataSourceInfo` object for each resource manager.

1. Create a `.java` file in the folder that contains your application code to provide values for the MicroTx client library properties.

   The following example provides the definition for the `DataSourceInfo` class in the `oracle.tmm.jta.common.DataSourceInfo` package with sample values.

   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. When you use multiple resource managers, you must specify values for the following MicroTx client library properties in a `.java` while initializing each data source.

   ```
   public class DataSourceInfo {
      String resourceManagerId = ORCL1-8976-9776-9873; //maps to the 
   oracle.tmm.xa.Rmid property
      String dataSourceName = creditDataSource; // name of the data 
   source
      boolean XaSupport = true; // maps to the oracle.tmm.xa.XaSupport 
   property
      boolean LLRSupport = false; // maps to the 
   oracle.tmm.xa.LLRSupport property
      boolean LRCSupport = false; // maps to the 
   oracle.tmm.xa.LRCSupport property
   }
   ```

   Note down the value you provide for the `dataSourceName`, as you will need to provide this name later when you inject a `connection` object for an XA-compliant resource manager.

   For details about each property and other optional properties, see Configure Library Properties.

2. Include the MicroTx library as a maven dependency in the application's `pom.xml` file. The following sample code is for the 22.3 release. Provide the correct version, based on the release that you want to use.

   ```
   <dependency>
        <groupId>com.oracle.tmm.jta</groupId>
        <artifactId>TmmLib</artifactId>
        <version>22.3</version>
   </dependency>
   ```

3. 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 xaDS, DataSourceInfo
   creditDataSource)
   ```

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

   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
   ```

   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,
   DataSourceInfo 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.

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 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.

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 = 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.

8. Only for a participant microservice based on Spring Boot, register the `XAResourceCallbacks` (prepare/commit/rollback) and various filters as:

   ```
   @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.

9. Save the changes.

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

8.6.2.3 Configure Java App with a Non-XA JDBC Resource

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

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

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. When you use multiple resource managers, you must specify values for certain MicroTx client library properties in a `.java` while initializing a data source and other values in a `.properties` file for the 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`.

   - If you are using a single resource manager with your application, configure property values for the MicroTx client library in the following way.
     - 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
       ```

   - If you are using a multiple resource managers with your application, configure property values for the MicroTx client library in the following way. Create a `.java` file in the folder that contains your application code to provide values for the MicroTx client library properties listed below.

The following example provides the definition for the `DataSourceInfo` class in the `oracle.tmm.jta.common.DataSourceInfo` package with sample values.

```
public class DataSourceInfo {
   String resourceManagerId = ORCL1-8976-9776-9873; //maps to
the oracle.tmm.xa.Rmid property
   String dataSourceName = creditDataSource; // name of the data
source
   boolean XaSupport = false; // maps to the
oracle.tmm.xa.XaSupport property
   boolean LLRSupport = false; // maps to the
oracle.tmm.xa.LLRSupport property
   boolean LRCSupport = true; // maps to the
oracle.tmm.xa.LRCSupport property
}
```

2. Create a `.properties` file in the folder that contains your application code to provide values for the following MicroTx client library properties for the application. The following example provides sample values for other MicroTx client library properties.

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

   For details about each property and other optional properties, see Configure Library Properties.

3. Include the MicroTx library as a maven dependency in the application's `pom.xml` file. The following sample code is for the 22.3 release. Provide the correct version, based on the release that you want to use.

```
<dependency>
     <groupId>com.oracle.tmm.jta</groupId>
     <artifactId>TmmLib</artifactId>
     <version>22.3</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 dataSource, DataSourceInfo
  dataSourceInfo)
  ```

  Where, `dataSourceInfo` is the object that you have created in the first step.

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.MysqlData
   Source");
   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 a participant microservice based on Spring Boot, register the XAResource callbacks, such as prepare, commit, rollback, and various filters as:

    ```
    @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.

8.6.2.4 Configure Java App with a Non-XA and Non-JDBC Resource

Use the information provided in this section to configure your Java 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 library as a maven dependency in the application's `pom.xml` file. The following sample code is for the 22.3 release. Provide the correct version, based on the release that you want to use.

   ```
   <dependency>
        <groupId>com.oracle.tmm.jta</groupId>
        <artifactId>TmmLib</artifactId>
        <version>22.3</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.

8.6.3 Configure JPA or Hibernate 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.

Configuring a JPA or Hibernate app as a transaction participant is similar to configuring a Java app as a transaction participant.

To configure a Java app as a transaction participant, you create a custom data source object and then pass this object to the MicroTx library. In your Java application code, a connection object was injected from the MicroTx library, and then the application code uses the injected object.

To configure a JPA or Hibernate app as a transaction participant, you create an entity manager factory object, and then pass this object to the MicroTx library. In your application code, the MicroTx library injects the entity manager factory object and your application code uses the injected object.

- Configure Hibernate or JPA App with an XA-Compliant Resource Manager
  Use the information provided in this section to configure your Hibernate or JPA applications as a participant when you use an XA-compliant resource manager.

8.6.3.1 Configure Hibernate or JPA App with an XA-Compliant Resource Manager

Use the information provided in this section to configure your Hibernate or JPA applications as a participant 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 library as a maven dependency in the application's `pom.xml` file. The following sample code is for the 22.3 release. Provide the correct version, based on the release that you want to use.

   ```
   <dependency>
        <groupId>com.oracle.tmm.jta</groupId>
        <artifactId>TmmLib</artifactId>
        <version>22.3</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 following example code shows how you can initialize the library in within the `XADataSourceConfig` class, create a custom data source named `ucpXADataSource`, 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.EnableTransactionManageme
nt;

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 = "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(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(HibernatePersistenceP
rovider.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, ucpXADataSource,
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, ucpXADataSource,
  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.
  - *`ucpXADataSource`* 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. 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 as shown in the following code sample.

  ```
  @Inject
  @TrmEntityManager(name = "ucpXADataSource")
  private EntityManager emf;
  ```

  Where, *`emf`* is the entity manager factory object and *`ucpXADataSource`* is the data source object that you have created in the previous step.

5. 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();
    }
```

6. Only for a participant microservice based on Spring Boot, register the `XAResource` callbacks (prepare/commit/rollback) and various filters as:

```
@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.

7. Save the changes.

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

8.7 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 library.

8.7.1 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.

   ```
   "dependencies": {
       "tmmlib-node": "file:tmmlib-node-<version>.tgz"
     }
   ```

2. Specify property values for the MicroTx 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 .

8.7.2 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 library.

- **Configure Node.js Apps with an XA-Compliant Resource Manager**
- Configure Node.js Apps with a Non-XA Resource

- 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.

8.7.2.1 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.

```
"dependencies": {
    "tmmlib-node": "file:tmmlib-node-<version>.tgz"
  }
```

2. Configure the property values for the MicroTx library. See Configure Library Properties.

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();
    }
}
```

8.7.2.2 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.

You can use a non-XA resource as a resource manager only for a transaction participant service that has a single replica. If the transaction participant service has multiple replicas, you can't use a non-XA resource.

1. Add the MicroTx library for Node.js as a dependency in the `package.json` file.

```
"dependencies": {
    "tmmlib-node": "file:tmmlib-node-<version>.tgz"
  }
```

2. Configure the property values for the MicroTx library. See Configure Library Properties.

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.

8.8 Develop ORDS App as Transaction Participant

This section provides the detailed steps to configure a database application as an XA participant in the context of deploying and running the Oracle Database sample application.

You can configure an Oracle Database application as a transaction participant in a transaction with MicroTx. The Oracle Database application, that you have built using Oracle Apex and Oracle REST Data Services (ORDS), is supported only as an XA transaction participant.

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 or in an Oracle *RAD* stack stack deployed in a Kubernetes cluster, or in an Oracle *RAD* stack deployed within a VM or a physical host. The Oracle *RAD* stack is an inclusive technology stack based on three core components: Oracle REST Data Services (ORDS), Oracle APEX, and Oracle Database.

- Prerequisites
- Run MicroTx Library for SQL
  The MicroTx library in PL/SQL 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.
- Build the ORDS App
- Run an XA Transaction
  Let's understand how to run an XA transaction by using the XA sample application as an example.

8.8.1 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 https://docs.oracle.com/en/database/oracle/application-express/21.1/aeutl/accessing-RESTful-services.html.
- Ensure that the ORDS service is available for the schema you have registered. For example, `http://localhost:50080/ords`. Log in to your APEX workspace using the user credentials of the schema.
- 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, and then add it to the database. You will need `sysdba` permissions to add an ACL. The following example shows a sample ACL. For more information about adding the required 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.

8.8.2 Run MicroTx Library for SQL

The MicroTx library in PL/SQL 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.

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.

   This file is located in the *`installation_directory`*`/otmm-`*`RELEASE`*`/samples/xa/plsql/lib` folder.

   This creates a set of PL/SQL functions and stored procedures.

8.8.3 Build the ORDS App

The `TmmStart` function enables the XA transaction to be coordinated by MicroTx. It makes a REST call to MicroTx to enlist the participant in the XA transaction and register the callback REST APIs.

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.

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.

Call the `TmmEnd` function after the business logic has been completely executed.

1. Create DDLs for tables and other database objects required for your application.
2. Add required DMLs to insert default data.
3. Create or define a new REST module for your application.
4. Create the required PL/SQL functions and stored procedures for your application.
5. 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;
/
```

8.8.4 Run an XA Transaction

Let's understand how to run an XA transaction by using the XA sample application as an example.

Code for the sample applications is available in the MicroTx installation bundle.

1. Run the sample application, `ordsapp.sql` file using SQL Developer or SQL Plus.

   This file is located in the `*installation_directory*/otmm-*RELEASE*/samples/xa/plsql/databaseapp` folder. Connect to the Oracle Database using the schema user that you have registered with ORDS. You can connect using SQL Developer or SQLPlus.

   This creates all database objects such as tables, a set of PL/SQL functions and stored procedures. Also, a REST module will be created along with all the REST APIs. The application is ready to serve REST API calls at this point.

2. Run the following commands to test the sample application. The calls to the `withdraw` and `deposit` REST APIs in the following sample code are executed locally within the application without being part of an XA transaction. Use these sample commands only to test that the sample application works as designed. In these sample commands, account is 222 and the port is 50080. Replace these values with information specific to your environment.

   a. Check balance for account 222.

   ```
   curl --location --request GET 'http://localhost:50080/ords/
   ordstest/accounts/222'
   ```

   b. Call the `withdraw` REST API for account 222 with an amount of 10.

   ```
   curl --location --request POST 'http://localhost:50080/ords/
   ordstest/accounts/222/withdraw?amount=10'
   ```

c. Check balance for account 222 to verify if the withdraw function was successful.

```
curl --location --request GET 'http://localhost:50080/ords/ordstest/
accounts/222'
```

d. Call the deposit REST API for account 222 with an amount of 10.

```
curl --location --request POST 'http://localhost:50080/ords/ordstest/
accounts/222/deposit?amount=10'
```

e. Check balance for account 222 to verify if the deposit function was successful. Use the correct account in place of 222 if you used some other account in the transfer request.

```
curl --location --request GET 'http://localhost:50080/ords/ordstest/
accounts/222'
```

3. Configure the ORDS application as an XA participant in the initiator application. To use this database application as a participant in an XA Transaction with MicroTx involving the other XA sample applications, make the changes as shown below.

```
//In the deployment descriptor for the accounts service,
//modify the env variable departmentTwoEndpoint value to the
//ORDS application URL "https://host:port/ords/schema/accounts".
            name: departmentTwoEndpoint
            value: https://host:port/ords/schema/accounts
```

   Where, *schema* is the schema that you have registered with ORDS.

4. Run an XA transaction. See Run an XA Transaction.

9

Develop Applications with LRA

The Transaction Manager for Microservices (MicroTx) library for Node.js provides the functionality to initiate a new LRA transaction or to participate in an existing LRA 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 LRA 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
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 LRA • Develop Node.js Apps with LRA
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 LRA 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

- Develop Java Apps with LRA
  Eclipse MicroProfile provides the LRA specification for Java applications.
- Configure Library Properties
  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 LRA
  The MicroTx library for Node.js provides the functionality to initiate a new LRA transaction or to participate in an existing LRA transaction. You must integrate this library into your Node.js application code.

9.1 Develop Java Apps with LRA

Eclipse MicroProfile provides the LRA specification for Java applications.

For more information, see https://download.eclipse.org/microprofile/microprofile-lra-1.0-M1/microprofile-lra-spec.html.

Helidon provides the implementation for LRA client specifications. For information, see https://helidon.io/docs/v2/#/mp/lra/01_introduction. For information about the implementation for applications, see https://danielkec.github.io/blog/helidon/lra/saga/2021/10/12/helidon-lra.html.

When the business logic of your application spawns across multiple API calls to complete a transaction

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.

9.2 Configure Library Properties

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 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. 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 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`.

For example,

```
oracle.tmm.TcsUrl = http://tmm-app:9000/api/v1
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 `application.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.

9.3 Develop Node.js Apps with LRA

The MicroTx library for Node.js provides the functionality to initiate a new LRA transaction or to participate in an existing LRA transaction. You must integrate this library into your Node.js application code.

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:tmmlib-node-RELEASE.tgz"
     }
   ```

2. 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');
   ```

3. 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';
   ```

4. Create a router object to handle requests in your program.

   Use the following code to create a router object named `flightSvcRouter`.

   ```
   const flightSvcRouter = Router();
   ```

5. Enter the URL of the MicroTx LRA 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
```

6. 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 LRA 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 LRA transaction.
   - `LRAType.REQUIRES_NEW` determines if the service participates in an existing LRA 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 LRA transaction, MicroTx adds a link header for all the outgoing requests from the REST API endpoints that you have specified.

7. 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.

8. Save your changes.

10 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 Java 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
- Develop Python Apps with TCC
  From the MicroTx release 22.3.2, MicroTx client libraries for Python applications provides the functionality to initiate a new TCC transaction or to participate in an existing TCC transaction.

10.1 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 Java 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

10.2 Configure Library Properties

Provide configuration information for the MicroTx client library properties. You must perform this step 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`: 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`: 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`.
- `server.port`: Enter the port over which you want to access the microservice. Create the required networking rules to permit inbound and outbound traffic on this port. For example, `8080`.

- `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.

  If the MicroTx coordinator is running inside a Docker container, in Ubuntu 20 or Docker Engine 20, with the network setting as `{--add-host host.docker.internal:host-gateway}`, then the callback URL is `http://host.docker.internal:{server.port}`.

  In other Docker environments, the structure of URL may vary depending on the operating system and its version.

For example,

```
oracle.tmm.TcsUrl = http://tmm-app:9000/api/v1
oracle.tmm.PropagateTraceHeaders = true
oracle.tmm.CallbackUrl = https://bookTicket-app:8081
server.port = 8081
```

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.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_PROPAGATE_TRACE_HEADERS = true
```

Note that the environment variables names are case-sensitive.

10.3 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.

10.4 Develop Java 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.

- Configure Java App as Transaction Initiator
- Configure Java App as Transaction Participant

10.4.1 Configure Java App as Transaction Initiator

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

1. Include the MicroTx library as a maven dependency in the application's `pom.xml` file. The following sample code is for the 22.3 release. Provide the correct version, based on the release that you want to use.

```
<dependency>
     <groupId>com.oracle.tmm.jta</groupId>
     <artifactId>TmmLib</artifactId>
     <version>22.3</version>
</dependency>
```

2. 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.

   ```
   import oracle.trm.tcc.annotation.TCC;
   @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.

10.4.2 Configure Java App as Transaction Participant

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

1. Include the MicroTx library as a maven dependency in the application's `pom.xml` file. The following sample code is for the 22.3 release. Provide the correct version, based on the release that you want to use.

   ```
   <dependency>
        <groupId>com.oracle.tmm.jta</groupId>
        <artifactId>TmmLib</artifactId>
        <version>22.3</version>
   </dependency>
   ```

2. 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.

```
import oracle.trm.tcc.annotation.TCC;
import javax.ws.rs.core.Application;

@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.
}
```

3. In the transaction participant application code, 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.

```
public class myTransactionParticipantApp extends Application {
    // Service code that is specific to the transaction participant
service.

    @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
      TccClient.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.

4. Save the changes.

Ensure that you make these changes in the code of all transaction participant services.

10.5 Develop Node.js Apps with TCC

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.

- Configure Node.js App as Transaction Initiator
- Configure Node.js App as Transaction Participant

10.5.1 Configure Node.js App as Transaction Initiator

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.

```
"dependencies": {
    "tmmlib-node": "file:tmmlib-node-<version>.tgz"
  }
```

2. 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');
   ```

3. 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';
   ```

4. Create a router object.

   Use the following code to create a router object named `svcRouter`.

   ```
   const svcRouter = Router();
   ```

5. 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.

10.5.2 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.

   ```
   "dependencies": {
       "tmmlib-node": "file:tmmlib-node-<version>.tgz"
     }
   ```

2. 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');
   ```

3. 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';
   ```

4. Create a router object.

   Use the following code to create a router object named `svcRouter2`.

   ```
   const svcRouter2 = Router();
   ```

5. 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.

6. 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
}));
```

7. 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.

10.6 Develop Python Apps with TCC

From the MicroTx release 22.3.2, 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.

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.

Method	Description
`GetTCCId(incoming_request_headers)`	Gets details of the current TCC transaction ID.

Where, `incoming_request_headers` is a dictionary of key-value pairs.

- Configure Python App as Transaction Initiator
- Configure Python App as Transaction Participant

10.6.1 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.

10.6.2 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.

11 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.

- Run Tuxedo App on Linux Host
- Run Tuxedo App in Kubernetes Cluster

11.1 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.

- Prepare the Environment
  Ensure that network connectivity is there between the Tuxedo application and MicroTx.
- Install Patches
  Apply patches for Tuxedo and SALT on your Tuxedo host.
- Verify the Set Up
  To verify that you have set up the Tuxedo environment properly, download the sample Tuxedo application and run it in the Tuxedo environment.

11.1.1 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.

11.1.2 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:
   1. Download the patch 24574032. For details about downloading the patch, see Downloading Release Update Patches in *Database Client Installation Guide for Linux*.
   2. Unzip the patch file.

      ```
      unzip p33664689_122200_Linux-x86-64.zip
      ```

   3. 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:
   1. Ensure that your Tuxedo application is still shutdown.
   2. Unzip the `RP902.zip` file that was supplied with the sample files.

      ```
      unzip RP902.zip
      ```

   3. 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.

11.1.3 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. See 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.

11.2 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.

11.2.1 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.

11.2.2 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 `installation_directory/otmm-<`*`version`*`>/samples/xa/helmcharts/transfer/templates` folder, 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. See Install XA Sample Application. 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. See 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.

12
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.

12.1 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
```

12.2 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
   ```

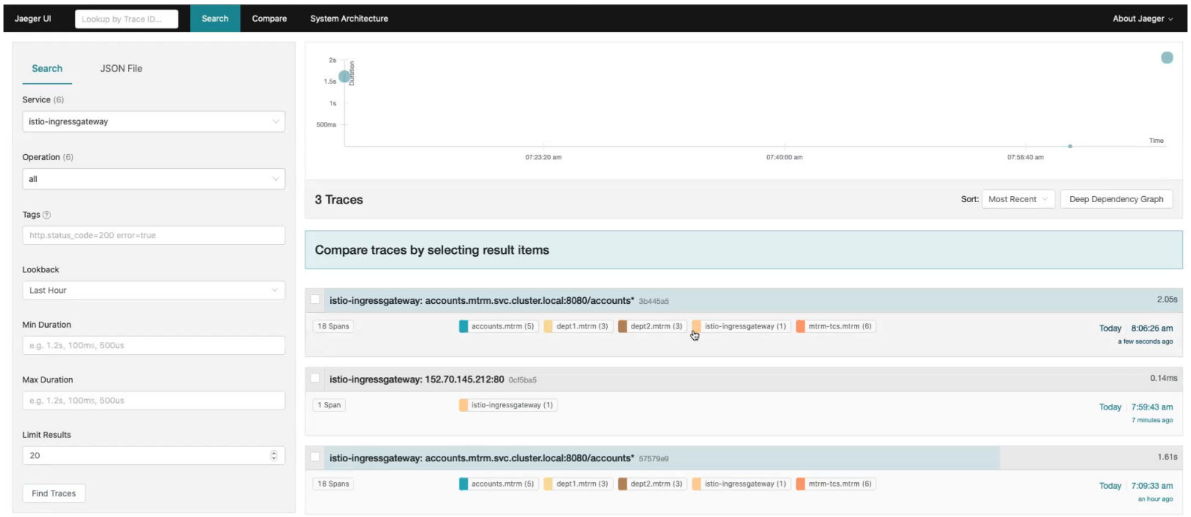
2. Perform a transaction using your application.

   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.

12.3 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
   ```

12.4 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.

`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/.

Name of the headers	Description

`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.

A

Manage Transaction Coordinator Using Helm

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.

- 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

A.1 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.

A.2 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.

A.3 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.

A.4 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
```

B
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

B.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.

B.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.

B.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.

B.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
   ```