

Using Transactions in Quarkus

Quarkus comes with a Transaction Manager and uses it to coordinate and expose transactions to your applications. Each extension dealing with persistence will integrate with it for you. And you will explicitly interact with transactions via CDI. This guide will walk you through all that.

Setting it up

You don't need to worry about setting it up most of the time as extensions needing it will simply add it as a dependency. Hibernate ORM for example will include the transaction manager and set it up properly.

You might need to add it as a dependency explicitly if you are using transactions directly without Hibernate ORM for example. Add the following to your `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-narayana-jta</artifactId>
</dependency>
```

Starting and stopping transactions: defining your boundaries

You can define your transaction boundaries the easy way, or the less easy way :)

Declarative approach

The easiest way to define your transaction boundaries is to use the `@Transactional` annotation on your entry method (`javax.transaction.Transactional`).

```

@ApplicationScoped
public class SantaClausService {

    @Inject ChildDAO childDAO;
    @Inject SantaClausDAO santaDAO;

    @Transactional ①
    public void getAGiftFromSanta(Child child, String
giftDescription) {
        // some transaction work
        Gift gift = childDAO.addToGiftList(child, giftDescription);
        if (gift == null) {
            throw new OMGGiftNotRecognizedException(); ②
        }
        else {
            santaDAO.addToSantaTodoList(gift);
        }
    }
}

```

① This annotation defines your transaction boundaries and will wrap this call within a transaction.

② A `RuntimeException` crossing the transaction boundaries will rollback the transaction.

`@Transactional` can be used to control transaction boundaries on any CDI bean at the method level or at the class level to ensure every method is transactional. That includes REST endpoints.

You can control whether and how the transaction is started with parameters on `@Transactional`:

- `@Transactional(REQUIRED)` (default): starts a transaction if none was started, stays with the existing one otherwise.
- `@Transactional(REQUIRES_NEW)`: starts a transaction if none was started ; if an existing one was started, suspends it and starts a new one for the boundary of that method.
- `@Transactional(MANDATORY)`: fails if no transaction was started ; works within the existing transaction otherwise.
- `@Transactional(SUPPORTS)`: if a transaction was started, joins it ; otherwise works with no transaction.
- `@Transactional(NOT_SUPPORTED)`: if a transaction was started, suspends it and works with no transaction for the boundary of the method ; otherwise works with no transaction.
- `@Transactional(NEVER)`: if a transaction was started, raises an exception ; otherwise works with no transaction.

`REQUIRED` or `NOT_SUPPORTED` are probably the most useful ones. This is how you decide whether a method is to be running within or outside a transaction. Make sure to check the JavaDoc for the precise semantic.

The transaction context is propagated to all calls nested in the `@Transactional` method as you

would expect (in this example `childDAO.addToGiftList()` and `santaDAO.addToSantaTodoList()`). The transaction will commit unless a runtime exception crosses the method boundary. You can override whether an exception forces the rollback or not by using `@Transactional(dontRollbackOn=SomeException.class)` (or `rollbackOn`).

You can also programmatically ask for a transaction to be marked for rollback. Inject a `TransactionManager` for this.

```
@ApplicationScoped
public class SantaClausService {

    @Inject TransactionManager tm; ❶
    @Inject ChildDAO childDAO;
    @Inject SantaClausDAO santaDAO;

    @Transactional
    public void getAGiftFromSanta(Child child, String
giftDescription) throws Exception {
        // some transaction work
        Gift gift = childDAO.addToGiftList(child, giftDescription);
        if (gift == null) {
            tm.setRollbackOnly(); ❷
        }
        else {
            santaDAO.addToSantaTodoList(gift);
        }
    }
}
```

❶ Inject the `TransactionManager` to be able to activate `setRollbackOnly` semantic.

❷ Programmatically decide to set the transaction for rollback.

Transaction Configuration

Extended configuration of the transaction is possible with the use of the `@TransactionConfiguration` annotation that is set in addition to the standard `@Transactional` annotation on your entry method or at the class level.

The `@TransactionConfiguration` annotation allows to set a timeout property, in seconds, that applies to transactions created within the annotated method.

This annotation may only be placed on the top level method delineating the transaction. Annotated nested methods once a transaction has started will throw an exception.

If defined on a class, it is equivalent to defining it on all the methods of the class marked with `@Transactional`. The configuration defined on a method takes precedence over the configuration defined on a class.

Reactive extensions

If your `@Transactional`-annotated method returns a reactive value, such as:

- `CompletionStage` (from the JDK)
- `Publisher` (from Reactive-Streams)
- Any type which can be converted to one of the two previous types using Reactive Type Converters

then the behaviour is a bit different, because the transaction will not be terminated until the returned reactive value is terminated. In effect, the returned reactive value will be listened to and if it terminates exceptionally the transaction will be marked for rollback, and will be committed or rolled-back only at termination of the reactive value.

This allows your reactive methods to keep on working on the transaction asynchronously until their work is really done, and not just until the reactive method returns.

If you need to propagate your transaction context across your reactive pipeline, please see the [Context Propagation guide](#).

API approach

The less easy way is to inject a `UserTransaction` and use the various transaction demarcation methods.

```
@ApplicationScoped
public class SantaClausService {

    @Inject ChildDAO childDAO
    @Inject SantaClausDAO santaDAO
    @Inject UserTransaction transaction

    public void getAGiftFromSanta(Child child, String
giftDescription) throws Exception {
        // some transaction work
        try {
            transaction.begin();
            Gift gift = childDAO.addToGiftList(child,
giftDescription);
            santaDAO.addToSantaTodoList(gift);
            transaction.commit();
        }
        catch(SomeException e) {
            // do something on Tx failure
            transaction.rollback();
        }
    }
}
```



You cannot use `UserTransaction` in a method having a transaction started by a `@Transactional` call.

Configuring the transaction timeout

You can configure the default transaction timeout, the timeout that applies to all transactions managed by the transaction manager, via the property `quarkus.transaction-manager.default-transaction-timeout`, specified as a duration.



The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the [Duration#parse\(\) javadoc](#).

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, `PT` is implicitly prepended to the value to obtain a standard `java.time.Duration` format.

The default value is 60 seconds.

Configuring transaction node name identifier

Narayana, as the underlying transaction manager, has a concept of a unique node identifier. This is important if you consider using XA transactions that involve multiple resources.

The node name identifier plays a crucial part in the identification of a transaction. The node name identifier is forged into the transaction id when the transaction is created. Based on the node name identifier, the transaction manager is capable of recognizing the XA transaction counterparts created in database or JMS broker. The identifier makes possible for the transaction manager to roll back the transaction counterparts during recovery.

The node name identifier needs to be unique per transaction manager deployment. And the node identifier needs to be stable over the transaction manager restarts.

The node name identifier may be configured via the property `quarkus.transaction-manager.node-name`.

Why always having a transaction manager?

Does it work everywhere I want to?

Yep, it works in your Quarkus application, in your IDE, in your tests, because all of these are Quarkus applications. JTA has some bad press for some people. I don't know why. Let's just say that this is not your grand'pa's JTA implementation. What we have is perfectly embeddable and lean.

Does it do 2 Phase Commit and slow down my app?

No, this is an old folk tale. Let's assume it essentially comes for free and let you scale to more

complex cases involving several datasources as needed.

I don't need transaction when I do read only operations, it's faster.

Wrong.

First off, just disable the transaction by marking your transaction boundary with `@Transactional(NOT_SUPPORTED)` (or `NEVER` or `SUPPORTS` depending on the semantic you want).

Second, it's again fairy tale that not using transaction is faster. The answer is, it depends on your DB and how many SQL SELECTs you are making. No transaction means the DB does have a single operation transaction context anyways.

Third, when you do several SELECTs, it's better to wrap them in a single transaction because they will all be consistent with one another. Say your DB represents your car dashboard, you can see the number of kilometers remaining and the fuel gauge level. By reading it in one transaction, they will be consistent. If you read one and the other from two different transactions, then they can be inconsistent. It can be more dramatic if you read data related to rights and access management for example.

Why do you prefer JTA vs Hibernate's transaction management API

Managing the transactions manually via `entityManager.getTransaction().begin()` and friends lead to a butt ugly code with tons of try catch finally that people get wrong. Transactions are also about JMS and other database access, so one API makes more sense.

It's a mess because I don't know if my JPA persistence unit is using JTA or Resource-level Transaction

It's not a mess in Quarkus :) Resource-level was introduced to support JPA in a non managed environment. But Quarkus is both lean and a managed environment so we can safely always assume we are in JTA mode. The end result is that the difficulties of running Hibernate ORM + CDI + a transaction manager in Java SE mode are solved by Quarkus.