Red Hat build of Quarkus 1.11

Managing JTA transactions with the Quarkus transaction manager

Last Updated: 2021-06-15

Legal Notice

Abstract

Use the Narayana JTA extension to manage transactions in your Quarkus application.

Table of Contents

PREFACE

As an application developer, you can use the Quarkus transaction manager to coordinate and expose JTA transactions to your applications.

Quarkus provides a transaction manager for coordinating JTA transactions across one or more resources. You can use the Quarkus transaction manager to control transaction boundaries in a declarative or in a programmatic way. You can also modify transactions and configure the transaction timeout. This functionality is provided by the **quarkus-narayana-jta** extension.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our technical content and encourage you to tell us what you think. If you'd like to add comments, provide insights, correct a typo, or even ask a question, you can do so directly in the documentation.

NOTE

You must have a Red Hat account and be logged in to the customer portal.

To submit documentation feedback from the customer portal, do the following:

1. Select the **Multi-page HTML** format.
2. Click the **Feedback** button at the top-right of the document.
3. Highlight the section of text where you want to provide feedback.
4. Click the **Add Feedback** dialog next to your highlighted text.
5. Enter your feedback in the text box on the right of the page and then click **Submit**.

We automatically create a tracking issue each time you submit feedback. Open the link that is displayed after you click **Submit** and start watching the issue or add more comments.

Thank you for the valuable feedback.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. PREREQUISITES

- Have OpenJDK (JDK) 11 installed and the **JAVA_HOME** environment variable specifies the location of the Java SDK.
    - Log in to the Red Hat Customer Portal to download Red Hat build of Open JDK from the Software Downloads page.
- Have Apache Maven 3.8.1 or higher installed.
    - Download Maven from the Apache Maven Project website.
- Have a Quarkus Maven project.
    - For information on how to create Quarkus applications with Maven, see Developing and compiling your Quarkus applications with Apache Maven.

CHAPTER 2. THE NARAYANA JTA TRANSACTION MANAGER AND QUARKUS

The Narayana JTA transaction manager lets you coordinate and expose JTA transactions to your Quarkus applications. You can include the **quarkus-narayana-jta** extension as a dependency to your project's **pom.xml** file and manage JTA transactions via annotations that are defined in the the **javax.transaction** package or via the Context and dependency injection (CDI).

The following table shows the most common Java Transaction APIs (JTA) annotations. .Java Transaction APIs (JTA) annotations:

Annotation	Description
@Transactional	Provides the ability to control transaction boundaries on any CDI beans at the method level or class level
@TransactionScoped	Provides the ability to specify a standard CDI scope to define bean instances whose life cycle is scoped to the currently active transaction

NOTE

You can set attributes on the **@Transactional** annotation to control how the transaction starts. You can apply the **@Transactional** annotation with attributes to individual methods or to the entire bean.

Additional resources

- Narayana community site
- API documentation for the **javax.transaction** package
- API documentation of the Transactional annotation
- API documentation of the Transactional annotation attributes

CHAPTER 3. INSTALLING THE QUARKUS NARAYANA JTA EXTENSION

You need to add the **quarkus-narayana-jta** extension as a dependency to your Quarkus project. If you are using Hibernate ORM, the **quarkus-narayana-jta** extension is already present in your project.

Prerequisites

- Have a Quarkus Maven project.

Procedure

1. Navigate to the root directory of your project.

```
cd <directory_name>
```

2. Use one of the following methods to add the **quarkus-narayana-jta** extension to your Quarkus project:

   a. Add the **quarkus-narayana-jta** extension to your **pom.xml** file:

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

   b. Add the **quarkus-narayana-jta** extension using the command line:

```
./mvnw quarkus:add-extension -Dextensions="narayana-jta"
```

CHAPTER 4. MANAGING JTA TRANSACTIONS DECLARATIVELY USING THE ANNOTATIONS

You can let the container demarcate transaction boundaries by automatically beginning and committing JTA transactions based on annotations. The following chapters demonstrate how you can manage JTA transactions and define transaction boundaries using the **@Transactional** annotation.

4.1. DEFINING TRANSACTION BOUNDARIES DECLARATIVELY

You can use **@Transactional** to control transaction boundaries on any CDI bean at the method level or at the class level to ensure that every method is transactional. This also applies to REST endpoints.

Procedure

- Define the scope of the transaction with the **@Transactional** annotation on the entry method:

  Example src/main/java/org/acme/SantaClauseService.java

```
import javax.inject.Inject;
import javax.enterprise.context.ApplicationScoped;
import 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) **@Transactional** annotation defines your transaction boundaries and wraps this call within a transaction.

(2) When a **RuntimeException** crosses the transaction boundaries, the transaction manager rolls back the transaction.

4.2. CONFIGURING A TRANSACTION FOR ROLLBACK DECLARATIVELY

Exceptions caused by system-level faults mark the transactions for rollback and abort the transaction immediately. You can override the default behavior using the **@Transactional(dontRollbackOn=SomeException.class)** or the **rollbackOn** attribute.

Prerequisites

- Have a Quarkus Maven project.

Procedure

- Use the **@Transactional(dontRollbackOn=SomeException.class)** to specify an exception that does not roll back the transaction:

Example src/main/java/org/acme/SantaClauseService.java

```
import javax.inject.Inject;
import javax.enterprise.context.ApplicationScoped;
import javax.transaction.Transactional;

@ApplicationScoped
public class SantaClausService {

    @Inject ChildDAO childDAO;
    @Inject SantaClausDAO santaDAO;

    @Transactional(dontRollbackOn=NonCriticalRuntimeException.class)
    public void getAGiftFromSanta(Child child, String giftDescription) throws Exception {
        Gift gift = childDAO.addToGiftList(child);

        // might throw a NonCriticalRuntimeException
        gift.setDescription(giftDescription);

        santaDAO.addToSantaTodoList(gift);
    }
}
```

In this example, the transaction context is propagated to all calls nested in the **@Transactional** method (**childDAO.addToGiftList()** and **santaDAO.addToSantaTodoList()**). The transaction commits unless a runtime exception crosses the method boundary.

4.3. CONFIGURING A TRANSACTION TIMEOUT DECLARATIVELY

Use the **@TransactionConfiguration** annotation in addition to the **@Transactional** annotation to specify the timeout in seconds. You can place the **@TransactionConfiguration** annotation only on the top-level method that delineates the transaction.

Procedure

- Use the **timeout** property of the **@TransactionConfiguration** to set the timeout in seconds:

```
import javax.transaction.Transactional;

@Transactional
```

```
@TransactionConfiguration(timeout=40)
public void getAGiftFromSanta(Child child, String giftDescription) {...}
```

NOTE

The configuration defined on a method takes precedence over the configuration defined on a class. When you define **@TransactionConfiguration** on a class, it is equivalent to defining it on all the methods of the class that are marked with **@Transactional**.

4.4. METHODS RETURNING REACTIVE VALUES

If a method annotated with **@Transactional** returns a reactive value it does not terminate the transaction until the returned reactive value is terminated. The transaction is marked for rollback when the reactive value terminates with an exception, otherwise the transaction is committed.

Additional resources

- Context Propagation guide

CHAPTER 5. MANAGING JTA TRANSACTIONS PROGRAMMATICALLY USING THE API APPROACH

You can manage transaction boundaries programmatically by injecting **UserTransaction**. The following chapters demonstrate how you can manage JTA transactions and define transaction boundaries using the API approach.

5.1. DEFINING TRANSACTION BOUNDARIES USING THE API APPROACH

You can inject a **UserTransaction** and manage the transaction boundaries by calling its **begin()**, **commit()** and **rollback()** methods.

Procedure

1. Inject the **UserTransaction** interface:

   src/main/java/org/acme/SantaClauseService.java

```
@ApplicationScoped
public class SantaClausService {

    @Inject ChildDAO childDAO;
    @Inject SantaClausDAO santaDAO;
    @Inject UserTransaction transaction;
}
```

2. Use the transaction demarcation methods to control the transaction:

   src/main/java/org/acme/SantaClauseService.java

```
import javax.transaction.Transactional;
import javax.inject.Inject;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;

@ApplicationScoped
public class SantaClausService {

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

    public void getAGiftFromSanta(Child child, String giftDescription) {
        // some transaction work
        try {
            transaction.begin(); (1)
            Gift gift = childDAO.addToGiftList(child, giftDescription);
            santaDAO.addToSantaTodoList(gift);
            transaction.commit();
        }
        catch(SomeException e) {
            // do something on Tx failure
```

```
            transaction.rollback(); 2
        }
    }
}
```

1. Place your transaction code between the **transaction.begin()** and the **transaction.commit()**.
2. Aborts the transaction immediately.

NOTE

You cannot use **UserTransaction** in a method where a transaction starts by a **@Transactional** call.

5.2. CONFIGURING A TRANSACTION FOR ROLLBACK USING THE API APPROACH

Exceptions caused by system-level faults mark the transactions for rollback. You can mark the transaction for rollback programmatically by injecting **TransactionManager**.

Procedure

1. Inject the **TransactionManager** and set the transaction for rollback with **setRollbackOnly**:
   In this example, the transaction context is propagated to all calls nested in the **@Transactional** method (**childDAO.addToGiftList()** and **santaDAO.addToSantaTodoList()**). The transaction manager commits the transaction unless a runtime exception crosses the method boundary.

   Example src/main/java/org/acme/SantaClausService.java

```
import javax.transaction.Transactional;
import javax.inject.Inject;
import javax.transaction.SystemException;
import javax.transaction.UserTransaction;

@ApplicationScoped
public class SantaClausService {

    @Inject TransactionManager tm; 1
    @Inject ChildDAO childDAO;
    @Inject SantaClausDAO santaDAO;

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

```
        }
    }
}
```

1. Inject the **TransactionManager** to be able to activate **setRollbackOnly** semantic.
2. Programmatically decide when to roll back the transaction.

CHAPTER 6. OVERWRITING THE DEFAULT TRANSACTION TIMEOUT

You can overwrite the transaction timeout by setting the value for the **quarkus.transaction-manager.default-transaction-timeout** property in your **application.properties** file. The default timeout for all transactions managed by the transaction manager is 60 seconds. If the transaction is not resolved within the timeout, the transaction manager automatically rolls it back.

Procedure

- Set the **<duration>** for the **quarkus.transaction-manager.default-transaction-timeout** property in your **application.properties** file:

  ```
  quarkus.transaction-manager.default-transaction-timeout=<duration>
  ```

  You can set the **<duration>** time in seconds or use the standard **java.time.Duration format**. For example, to set the timeout to 2 minutes, enter **quarkus.transaction-manager.default-transaction-timeout=PT2M**.

Additional resources

- API documentation for **Duration#parse()**

CHAPTER 7. CONFIGURING THE TRANSACTION NODE NAME IDENTIFIER FOR XA TRANSACTIONS

You can set a unique node identifier for XA transactions that have multiple resources. When you create a transaction the node name identifier becomes part of the transaction ID. The identifier allows the transaction manager to recognize the XA transaction counterparts created in a database or by a JMS broker. The transaction manager can roll back the transaction counterparts during recovery.

Procedure

- Set a value for the **quarkus.transaction-manager.node-name** property in your **application.properties** file:

```
quarkus.transaction-manager.node-name=<unique_id>
```

NOTE

Make sure to set a unique node name identifier for each deployment of transaction manager. The node identifier must be stable over the transaction manager restarts.

CHAPTER 8. OVERVIEW OF QUARKUS TRANSACTION CONFIGURATION PROPERTIES

The following table lists some of the configuration properties that you can use to configure the transaction management.

Table 8.1. Table Quarkus transaction configuration properties and their default values:

Property	Description	Default
quarkus.datasource.jdbc.transactions	Lets you use regular JDBC transactions, XA, or disable all transactional capabilities (**enabled**, **xa**, **disabled**)	enabled
quarkus.datasource.jdbc.transaction-isolation-level	The transaction isolation level (**undefined**, **none**, **read-uncommitted**, **read-committed**, **repeatable-read**, **serializable**)	
quarkus.transaction-manager.default-transaction-timeout	The timeout for all transactions managed by the transaction manager	60 seconds
quarkus.transaction-manager.node-name	The node name identifier	

Revised on 2021-06-15 14:51:53 UTC