(/)

Introducción a las transacciones en Java y Spring

Última modificación: 4 de agosto de 2020

por Kumar Chandrakant (https://www.baeldung.com/author/kumar-chandrakant/)
(https://www.baeldung.com/author/kumar-chandrakant/)

Java (https://www.baeldung.com/category/java/) +

Actas (https://www.baeldung.com/tag/transactions/)

1. Introducción

En este tutorial, entenderemos qué se entiende por transacciones en Java. De ese modo, entenderemos cómo realizar transacciones locales de recursos y transacciones globales. Esto también nos permitirá explorar diferentes formas de administrar transacciones en Java y Spring.

2. ¿Qué es una transacción?

Las transacciones en Java, como en general, se refieren a una serie de acciones que deben completarse con éxito (/transactions-intro) . Por lo tanto, si una o más acciones fallan, todas las demás acciones deben retroceder dejando el estado de la aplicación sin cambios . Esto es necesario para garantizar que la integridad del estado de la aplicación nunca se vea comprometida.

Además, estas transacciones pueden involucrar uno o más recursos como base de datos, cola de mensajes, dando lugar a diferentes formas de realizar acciones bajo una transacción. Estos incluyen la realización de transacciones de recursos locales con recursos individuales. Alternativamente, varios recursos pueden participar en una transacción global.

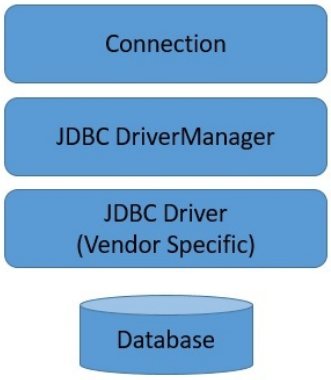
3. Transacciones locales de recursos

Primero exploraremos cómo podemos usar transacciones en Java mientras trabajamos con recursos individuales. Aquí, podemos tener múltiples acciones individuales que realizamos con un recurso como una base de datos . Pero, podemos querer que sucedan como un todo unificado, como en una unidad de trabajo indivisible. En otras palabras, queremos que estas acciones se realicen en una sola transacción.

En Java, tenemos varias formas de acceder y operar en un recurso como una base de datos. Por lo tanto, la forma en que manejamos las transacciones tampoco es la misma. En esta sección, encontraremos cómo podemos usar transacciones con algunas de estas bibliotecas en Java que se usan con bastante frecuencia.

3.1. JDBC

Java Database Connectivity (JDBC) (https://docs.oracle.com/javase/tutorial/jdbc/overview/index.html) es la API en Java que define cómo acceder a las bases de datos en Java . Los diferentes proveedores de bases de datos proporcionan controladores JDBC para conectarse a la base de datos de manera independiente del proveedor. Entonces, recuperamos una *conexión* de un controlador para realizar diferentes operaciones en la base de datos:



(/wp-content/uploads/2020/08/JDBC-Architecture.jpg)

JDBC nos brinda las opciones para ejecutar declaraciones en una transacción. El comportamiento predeterminado de una *conexión* es la confirmación automática . Para aclarar, lo que esto significa es que cada declaración se trata como una transacción y se confirma automáticamente justo después de la ejecución.

Sin embargo, si deseamos agrupar varias declaraciones en una sola transacción, esto también es posible:

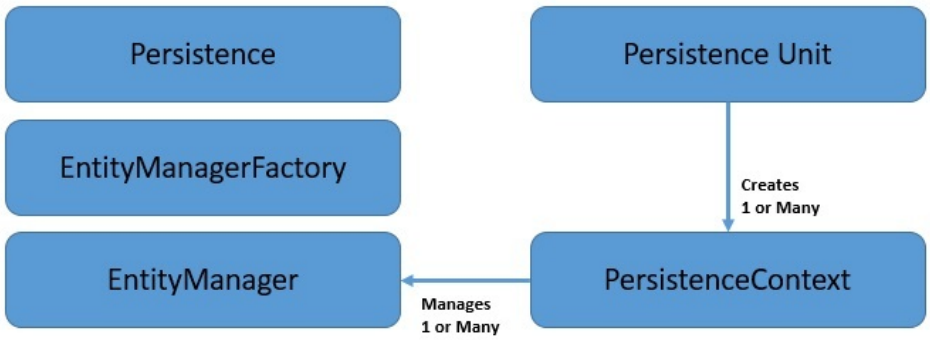
```
Connection connection = DriverManager.getConnection(CONNECTION_URL, USER, PASSWORD);
try {
    connection.setAutoCommit(false);
    PreparedStatement firstStatement = connection .prepareStatement("firstQuery");
    firstStatement.executeUpdate();
    PreparedStatement secondStatement = connection .prepareStatement("secondQuery");
    secondStatement.executeUpdate();
    connection.commit();
} catch (Exception e) {
    connection.rollback();
}
```

Aquí, hemos desactivado el modo de confirmación automática de *Connection* . Por lo tanto, podemos definir manualmente el límite de la transacción y realizar una *confirmación* o una *reversión* . JDBC también nos permite establecer un punto de *guardado* que nos brinda más control sobre cuánto revertir.

3.2. JPA

La API de persistencia de Java (JPA) (https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html) es una especificación en Java que se puede utilizar para cerrar la brecha entre los modelos de dominio orientados a objetos y los sistemas de bases de datos relacionales . Por lo tanto, existen varias implementaciones de JPA disponibles de terceros como Hibernate, EclipseLink e iBatis.

En JPA, podemos definir clases regulares como una *entidad* que les proporciona una identidad persistente. La clase *EntityManager* proporciona la interfaz necesaria para trabajar con múltiples entidades dentro de un contexto de persistencia . El contexto de persistencia se puede considerar como una caché de primer nivel donde se administran las entidades:



(/wp-content/uploads/2020/08/JPA-Architecture.jpg)
Arquitectura JPA

El contexto de persistencia aquí puede ser de dos tipos, de alcance de transacción o de alcance extendido. Un contexto de persistencia con alcance de transacción está vinculado a una sola transacción. Mientras que el contexto de persistencia de alcance extendido puede abarcar varias transacciones. El alcance predeterminado de un contexto de persistencia es el alcance de la transacción .

Veamos cómo podemos crear un *EntityManager* y definir un límite de transacción manualmente:

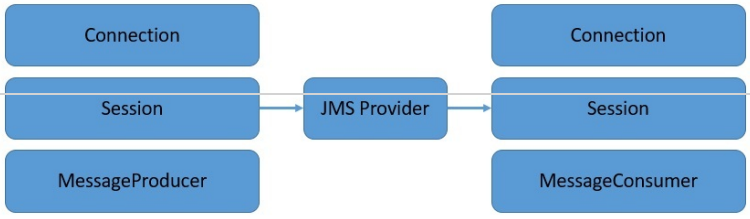
```
EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("jpa-example");
EntityManager entityManager = entityManagerFactory.createEntityManager();
try {
    entityManager.getTransaction().begin();
    entityManager.persist(firstEntity);
    entityManager.persist(secondEntity);
    entityManager.getTransaction().commit();
} catch (Exceotion e) {
    entityManager.getTransaction().rollback();
}
```

Aquí, estamos creando un *EntityManager* desde *EntityManagerFactory* dentro del contexto de un contexto de persistencia con alcance de transacción. Luego, estamos definiendo el límite de la transacción con los métodos *begin* , *commit* y *rollback* .

3.3. JMS

Java Messaging Service (JMS) (https://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html) es una especificación en Java que permite que las aplicaciones se comuniquen de forma asíncrona mediante mensajes . La API nos permite crear, enviar, recibir y leer mensajes de una cola o tema. Hay varios servicios de mensajería que cumplen con las especificaciones de JMS, incluidos OpenMQ y ActiveMQ.

La API de JMS admite la agrupación de varias operaciones de envío o recepción en una única transacción. Sin embargo, por la naturaleza de la arquitectura de integración basada en mensajes, la producción y el consumo de un mensaje no pueden formar parte de la misma transacción . El alcance de la transacción permanece entre el cliente y el proveedor JMS:



(/wp-content/uploads/2020/08/JMS-Architecture.jpg)

JMS nos permite crear una *sesión a* partir de una *conexión* que obtenemos de un *ConnectionFactory* específico del proveedor . Tenemos la opción de crear una *sesión* que se transmita o no . Para no transacciones *Sesión* s , podemos definir, además, un modo apropiado reconocer también.

Veamos cómo podemos crear una *sesión* con transacción para enviar varios mensajes en una transacción:

```
ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(CONNECTION_URL);
Connection connection = = connectionFactory.createConnection();
connection.start();
try {
    Session session = connection.createSession(true, 0);
    Destination = destination = session.createTopic("TEST.FOO");
    MessageProducer producer = session.createProducer(destination);
    producer.send(firstMessage);
    producer.send(secondMessage);
    session.commit();
} catch (Exception e) {
    session.rollback();
}
```

Aquí, estamos creando un *MessageProducer* para el *destino* del tipo de tema. Obtenemos el *destino* de la *sesión* que creamos anteriormente. Además usamos *Session* para definir los límites de las transacciones usando los métodos *commit* y *rollback* .

4. Transacciones globales

Como vimos, las transacciones locales de recursos nos permiten realizar múltiples operaciones dentro de un solo recurso como un todo unificado. Pero, con bastante frecuencia, nos ocupamos de operaciones que abarcan varios recursos . Por ejemplo, operación en dos bases de datos diferentes o una base de datos y una cola de mensajes. Aquí, el soporte de transacciones locales dentro de los recursos no será suficiente para nosotros.

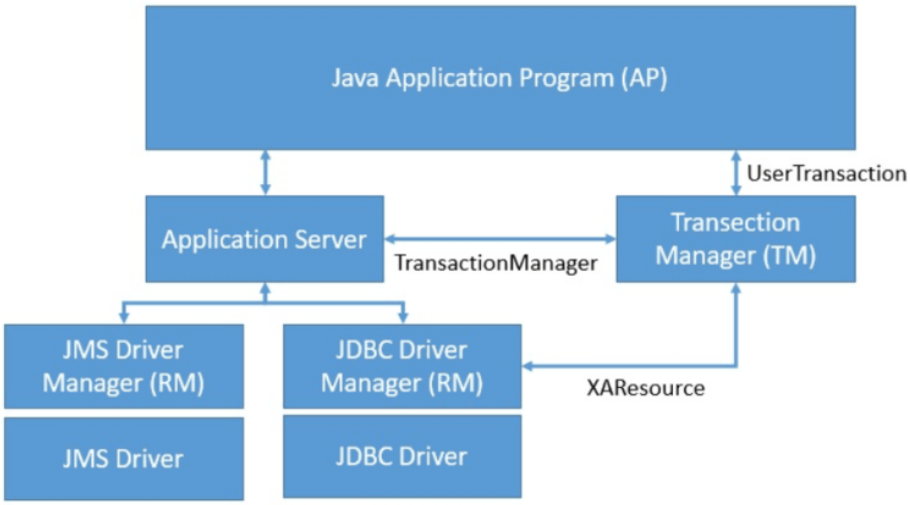
Lo que necesitamos en estos escenarios es un mecanismo global para demarcar transacciones que abarcan múltiples recursos participantes . Esto a menudo se conoce como transacciones distribuidas y hay especificaciones que se han propuesto para tratarlas de manera efectiva.

La Especificación XA es una de esas especificaciones que define a un administrador de transacciones para controlar las transacciones en múltiples recursos . Java tiene un soporte bastante maduro para transacciones distribuidas que se ajustan a la Especificación XA a través de los componentes JTA y JTS.

4.1. JTA

Java Transaction API (JTA) (https://www.oracle.com/technetwork/java/javaee/jta/index.html) es una API de Java Enterprise Edition desarrollada bajo el Proceso de comunidad de Java. Se permite que las aplicaciones Java y los servidores de aplicaciones para realizar transacciones distribuidas a través de recursos XA . JTA se modela en torno a la arquitectura XA, aprovechando el compromiso de dos fases.

JTA especifica interfaces Java estándar entre un administrador de transacciones y las otras partes en una transacción distribuida:



(/wp-content/uploads/2020/08/Screenshot-2020-04-25-at-06.53.12.png)

Entendamos algunas de las interfaces clave destacadas anteriormente:

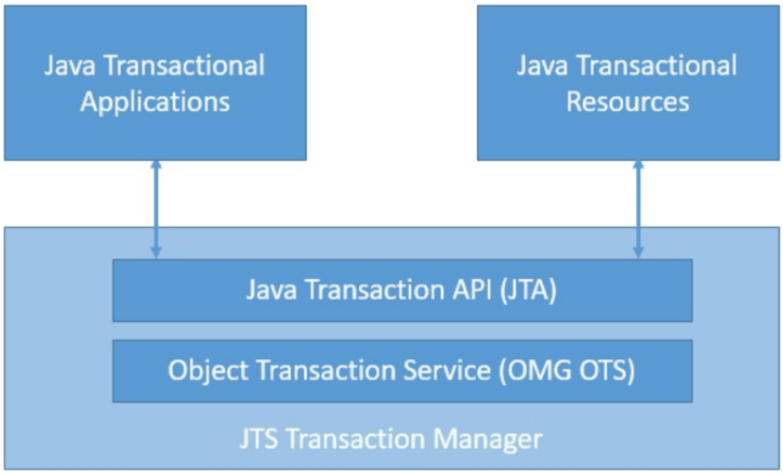
- *TransactionManager* : una interfaz que permite a un servidor de aplicaciones demarcar y controlar transacciones
- *UserTransaction* : esta interfaz permite que un programa de aplicación demarque y controle transacciones explícitamente

- *XAResource* : el propósito de esta interfaz es permitir que un administrador de transacciones trabaje con administradores de recursos para recursos compatibles con XA

4.2. JTS

Java Transaction Service (JTS) (https://download.oracle.com/otndocs/jcp/7309-jts-1.0-spec-oth-JSpec/) es una especificación para crear el administrador de transacciones que se asigna a la especificación OMG OTS . JTS utiliza las interfaces CORBA ORB / TS estándar y el Protocolo Inter-ORB de Internet (IIOP) para la propagación del contexto de la transacción entre los administradores de transacciones de JTS.

En un nivel alto, es compatible con la API de transacciones de Java (JTA). Un administrador de transacciones de JTS proporciona servicios de transacciones a las partes involucradas en una transacción distribuida:



(/wp-content/uploads/2020/08/Screenshot-2020-04-25-at-06.53.53.png)

Los servicios que JTS proporciona a una aplicación son en gran medida transparentes y, por lo tanto, es posible que ni siquiera los notemos en la arquitectura de la aplicación. JTS está diseñado alrededor de un servidor de aplicaciones que abstrae toda la semántica de transacciones de los programas de aplicación.

5. Gestión de transacciones de JTA

Ahora es el momento de comprender cómo podemos gestionar una transacción distribuida utilizando JTA. Las transacciones distribuidas no son soluciones triviales y, por lo tanto, también tienen implicaciones en los costos. Además, existen múltiples opciones entre las que podemos elegir para incluir JTA en nuestra aplicación . Por lo tanto, nuestra elección debe basarse en la arquitectura y las aspiraciones generales de la aplicación.

5.1. JTA en el servidor de aplicaciones

Como hemos visto anteriormente, la arquitectura JTA se basa en el servidor de aplicaciones para facilitar una serie de operaciones relacionadas con las transacciones . Uno de los servicios clave que confía en el servidor para proporcionar es un servicio de nombres a través de JNDI. Aquí es donde los recursos XA como las fuentes de datos se vinculan y se recuperan.

Aparte de esto, tenemos una opción en términos de cómo queremos administrar el límite de la transacción en nuestra aplicación. Esto da lugar a dos tipos de transacciones dentro del servidor de aplicaciones Java:

- Transacción gestionada por contenedor : como sugiere el nombre, aquí el servidor de aplicaciones establece el límite de la transacción . Esto simplifica el desarrollo de Enterprise Java Beans (EJB) ya que no incluye declaraciones relacionadas con la demarcación de transacciones y se basa

únicamente en el contenedor para hacerlo. Sin embargo, esto no proporciona suficiente flexibilidad para la aplicación.
- Transacción gestionada por bean : a diferencia de la transacción gestionada por contenedor, en una transacción gestionada por bean, los EJB contienen las declaraciones explícitas para definir la demarcación de la transacción . Esto proporciona un control preciso a la aplicación al marcar los límites de la transacción, aunque a costa de una mayor complejidad.

Uno de los principales inconvenientes de realizar transacciones en el contexto de un servidor de aplicaciones es que la aplicación se acopla estrechamente con el servidor . Esto tiene implicaciones con respecto a la capacidad de prueba, la capacidad de administración y la portabilidad de la aplicación. Esto es más profundo en la arquitectura de microservicios, donde el énfasis está más en el desarrollo de aplicaciones neutrales al servidor.

5.2. JTA independiente

Los problemas que discutimos en la última sección han proporcionado un gran impulso hacia la creación de soluciones para transacciones distribuidas que no dependen de un servidor de aplicaciones . Hay varias opciones disponibles para nosotros a este respecto, como usar el soporte de transacciones con Spring o usar un administrador de transacciones como Atomikos.

Veamos cómo podemos usar un administrador de transacciones como Atomikos (/java-atomikos) para facilitar una transacción distribuida con una base de datos y una cola de mensajes. Uno de los aspectos clave de una transacción distribuida es la inclusión y exclusión de los recursos participantes con el supervisor de transacciones . Atomikos se encarga de esto por nosotros. Todo lo que tenemos que hacer es usar abstracciones proporcionadas por Atomikos:


```
AtomikosDataSourceBean atomikosDataSourceBean = new AtomikosDataSourceBean();
atomikosDataSourceBean.setXaDataSourceClassName("com.mysql.cj.jdbc.MysqlXADataSource");
DataSource dataSource = atomikosDataSourceBean;
```

Aquí, estamos creando una instancia de *AtomikosDataSourceBean* y registrando el *XADataSource* específico del *proveedor* . A partir de ahora, podemos seguir usando esto como cualquier otra *fuente de datos* y obtener los beneficios de las transacciones distribuidas.

De manera similar, tenemos una abstracción para la cola de mensajes que se encarga de registrar el recurso XA específico del proveedor con el monitor de transacciones automáticamente:

```
AtomikosConnectionFactoryBean atomikosConnectionFactoryBean = new AtomikosConnectionFactoryBean();
atomikosConnectionFactoryBean.setXaConnectionFactory(new ActiveMQXAConnectionFactory());
ConnectionFactory connectionFactory = atomikosConnectionFactoryBean;
```

Aquí, estamos creando una instancia de *AtomikosConnectionFactoryBean* y registrando *XAConnectionFactory* de un proveedor JMS habilitado para XA. Después de esto, podemos continuar usando esto como una *ConnectionFactory* regular .

Ahora, Atomikos nos proporciona la última pieza del rompecabezas para unir todo, una instancia de *UserTransaction* :

```
UserTransaction userTransaction = new UserTransactionImp();
```

Ahora, estamos listos para crear una aplicación con transacciones distribuidas que abarquen nuestra base de datos y la cola de mensajes:

```
try {
    userTransaction.begin();

    java.sql.Connection dbConnection = dataSource.getConnection();
    PreparedStatement preparedStatement = dbConnection.prepareStatement(SQL_INSERT);
    preparedStatement.executeUpdate();

    javax.jms.Connection mbConnection = connectionFactory.createConnection();
    Session session = mbConnection.createSession(true, 0);
    Destination destination = session.createTopic("TEST.FOO");
    MessageProducer producer = session.createProducer(destination);
    producer.send(MESSAGE);

    userTransaction.commit();
} catch (Exception e) {
    userTransaction.rollback();
}
```

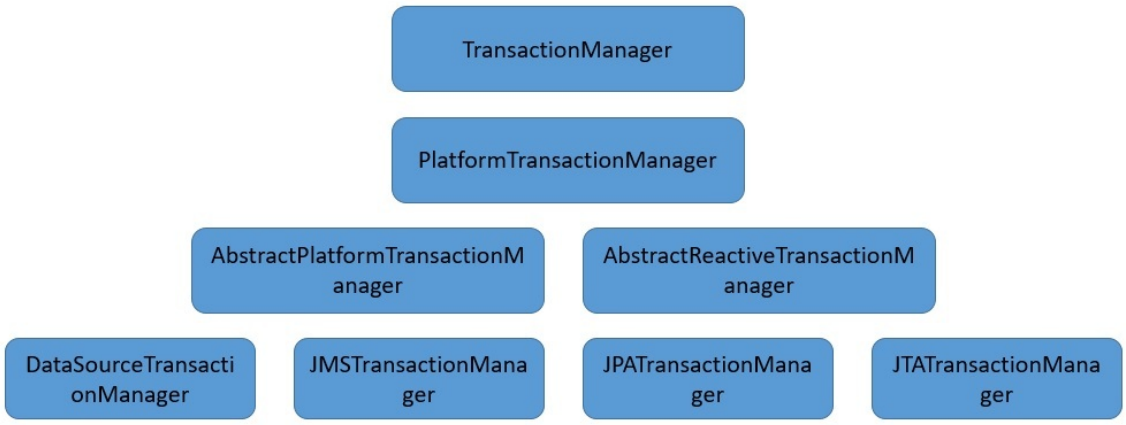
Aquí, estamos usando los métodos *begin* y *commit* en la clase *UserTransaction* para demarcar el límite de la transacción . Esto incluye guardar un registro en la base de datos y publicar un mensaje en la cola de mensajes.

6. Soporte de transacciones en Spring

Hemos visto que el manejo de transacciones es una tarea bastante complicada que incluye una gran cantidad de codificación y configuraciones estándar . Además, cada recurso tiene su propia forma de gestionar las transacciones locales. En Java, JTA nos abstrae de estas variaciones, pero además aporta detalles específicos del proveedor y la complejidad del servidor de aplicaciones.

La plataforma Spring nos proporciona una forma mucho más limpia de manejar transacciones, tanto transacciones de recursos locales como globales en Java. Esto, junto con los otros beneficios de Spring, crea un caso convincente para usar Spring para manejar transacciones. Además, es bastante fácil configurar y cambiar un administrador de transacciones con Spring, que puede ser proporcionado por el servidor o independiente.

Spring nos proporciona esta perfecta abstracción al crear un proxy para los métodos con código transaccional. El proxy administra el estado de la transacción en nombre del código con la ayuda de *TransactionManager* :



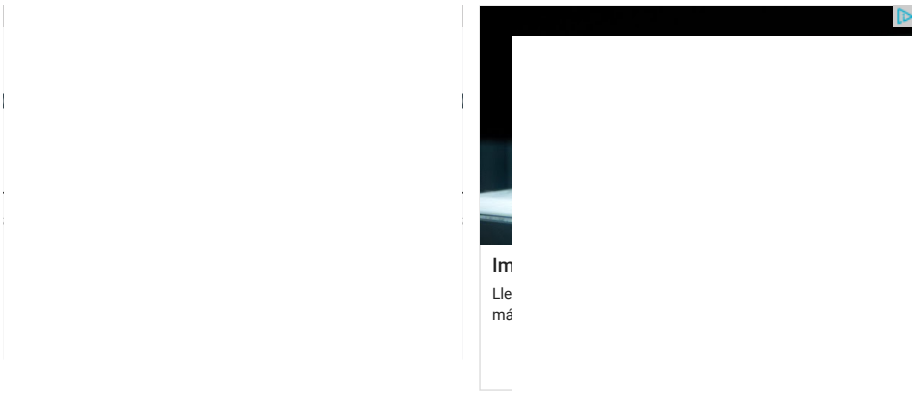
la interfaz central aquí es *PlatformTransactionManager,* que tiene varias implementaciones diferentes disponibles. Proporciona abstracciones sobre JDBC (DataSource), JMS, JPA, JTA y muchos otros recursos.

6.1. Configuraciones

Veamos cómo podemos configurar Spring para usar Atomikos como administrador de transacciones y brindar soporte transaccional para JPA y JMS . Comenzaremos definiendo un *PlatformTransactionManager* del tipo JTA:

```
@Bean
public PlatformTransactionManager platformTransactionManager() throws Throwable {
    return new JtaTransactionManager(
                userTransaction(), transactionManager());
}
```

Aquí, proporcionamos instancias de *UserTransaction* y *TransactionManager* a *JTATransactionManager* . Estas instancias las proporciona una biblioteca de administrador de transacciones como Atomikos:



```
@Bean
public UserTransaction userTransaction() {
    return new UserTransactionImp();
}

@Bean(initMethod = "init", destroyMethod = "close")
public TransactionManager transactionManager() {
    return new UserTransactionManager();
}
```

Las clases *UserTransactionImp* y *UserTransactionManager* son proporcionadas por Atomikos aquí.

Además, necesitamos definir el *JmsTemplete*, que es la clase principal que permite el acceso JMS sincrónico en Spring:

```
@Bean
public JmsTemplate jmsTemplate() throws Throwable {
    return new JmsTemplate(connectionFactory());
}
```

Aquí, *ConnectionFactory* es proporcionado por Atomikos donde habilita la transacción distribuida para *Connection* proporcionado por él:

```
@Bean(initMethod = "init", destroyMethod = "close")
public ConnectionFactory connectionFactory() {
    ActiveMQXAConnectionFactory activeMQXAConnectionFactory = new
ActiveMQXAConnectionFactory();
    activeMQXAConnectionFactory.setBrokerURL("tcp://localhost:61616");
    AtomikosConnectionFactoryBean atomikosConnectionFactoryBean = new
AtomikosConnectionFactoryBean();
    atomikosConnectionFactoryBean.setUniqueResourceName("xamq");
    atomikosConnectionFactoryBean.setLocalTransactionMode(false);
atomikosConnectionFactoryBean.setXaConnectionFactory(activeMQXAConnectionFactory);
    return atomikosConnectionFactoryBean;
}
```

Entonces, como podemos ver, aquí estamos empaquetando un *XAConnectionFactory* específico del proveedor JMS con *AtomikosConnectionFactoryBean* .

A continuación, necesitamos definir un *AbstractEntityManagerFactoryBean* que sea responsable de crear el bean *EntityManagerFactory* en Spring:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManager() throws SQLException {
    LocalContainerEntityManagerFactoryBean entityManager = new
LocalContainerEntityManagerFactoryBean();
    entityManager.setDataSource(dataSource());
    Properties properties = new Properties();
    properties.setProperty( "javax.persistence.transactionType", "jta");
    entityManager.setJpaProperties(properties);
    return entityManager;
}
```

Como antes, el *DataSource* que configuramos en *LocalContainerEntityManagerFactoryBean* aquí lo proporciona Atomikos con las transacciones distribuidas habilitadas:

```
@Bean(initMethod = "init", destroyMethod = "close")
public DataSource dataSource() throws SQLException {
    MysqlXADataSource mysqlXaDataSource = new MysqlXADataSource();
    mysqlXaDataSource.setUrl("jdbc:mysql://127.0.0.1:3306/test");
    AtomikosDataSourceBean xaDataSource = new AtomikosDataSourceBean();
    xaDataSource.setXaDataSource(mysqlXaDataSource);
    xaDataSource.setUniqueResourceName("xads");
    return xaDataSource;
}
```

Aquí nuevamente, estamos envolviendo el *XADataSource* específico del *proveedor* en *AtomikosDataSourceBean* .

6.2. Gestión de transacciones

Habiendo pasado por todas las configuraciones en la última sección, ¡debemos sentirnos bastante abrumados! Incluso podemos cuestionar los beneficios de usar Spring después de todo. Pero recuerde que toda esta configuración nos ha permitido abstraernos de la mayor parte del texto estándar específico del proveedor y nuestro código de aplicación real no necesita ser consciente de eso en absoluto.

Entonces, ahora estamos listos para explorar cómo usar transacciones en Spring donde pretendemos actualizar la base de datos y publicar mensajes. Spring nos brinda dos formas de lograr esto con sus propios beneficios para elegir. Entendamos cómo podemos hacer uso de ellos:

- Soporte declarativo

La forma más fácil de usar transacciones en Spring es con soporte declarativo. Aquí, tenemos una anotación de conveniencia disponible para ser aplicada en el método o incluso en la clase . Esto simplemente habilita la transacción global para nuestro código:

```
@PersistenceContext
EntityManager entityManager;

@Autowired
JmsTemplate jmsTemplate;

@Transactional(propagation = Propagation.REQUIRED)
public void process(ENTITY, MESSAGE) {
   entityManager.persist(ENTITY);
   jmsTemplate.convertAndSend(DESTINATION, MESSAGE);
}
```

El código simple anterior es suficiente para permitir una operación de guardar en la base de datos y una operación de publicación en la cola de mensajes dentro de una transacción JTA.

- Apoyo programático

Si bien el soporte declarativo es bastante elegante y simple, no nos ofrece el beneficio de controlar el límite de la transacción con mayor precisión . Por lo tanto, si tenemos cierta necesidad de lograrlo, Spring ofrece soporte programático para demarcar el límite de la transacción:

```
@Autowired
private PlatformTransactionManager transactionManager;

public void process(ENTITY, MESSAGE) {
    TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);
    transactionTemplate.executeWithoutResult(status -> {
        entityManager.persist(ENTITY);
        jmsTemplate.convertAndSend(DESTINATION, MESSAGE);
    });
}
```

Entonces, como podemos ver, tenemos que crear una *TransactionTemplate* con el *PlatformTransactionManager* disponible . Entonces podemos usar *TransactionTemplete* para procesar un montón de declaraciones dentro de una transacción global.

7. Reflexiones posteriores

Como hemos visto, el manejo de transacciones, particularmente aquellas que abarcan múltiples recursos, es complejo. Además, las transacciones se bloquean inherentemente, lo que es perjudicial para la latencia y el rendimiento. de una aplicación. Además, probar y mantener el código con transacciones distribuidas no es fácil, especialmente si la transacción depende del servidor de aplicaciones subyacente. Entonces, en general, ies mejor evitar las transacciones si podemos!

Pero eso está lejos de la realidad. En resumen, en las aplicaciones del mundo real, a menudo tenemos una necesidad legítima de transacciones. Aunque es posible repensar la arquitectura de la aplicación sin transacciones , puede que no siempre sea posible. Por lo tanto, debemos adoptar ciertas mejores prácticas cuando trabajamos con transacciones en Java para mejorar nuestras aplicaciones:

- Uno de los cambios fundamentales que deberíamos adoptar es utilizar administradores de transacciones independientes en lugar de los proporcionados por un servidor de aplicaciones . Esto solo puede simplificar enormemente nuestra aplicación. Además, es muy adecuado para la arquitectura de microservicios nativa de la nube.
- Además, una capa de abstracción como Spring puede ayudarnos a contener el impacto directo de los proveedores como JPA o JTA. Por lo tanto, esto puede permitirnos cambiar entre proveedores sin mucho impacto en nuestra lógica comercial. Además, nos quita las responsabilidades de bajo nivel de administrar el estado de la transacción.
- Por último, debemos tener cuidado al elegir el límite de la transacción en nuestro código . Dado que las transacciones se bloquean, siempre es mejor mantener el límite de la transacción lo más restringido posible. Si es necesario, deberíamos preferir el control programático sobre el declarativo para las transacciones.

8. Conclusión

En resumen, en este tutorial discutimos las transacciones en el contexto de Java. Pasamos por el soporte para transacciones locales de recursos individuales en Java para diferentes recursos. También analizamos las formas de lograr transacciones globales en Java.

Además, analizamos diferentes formas de administrar transacciones globales en Java. Además, entendimos cómo Spring nos facilita el uso de transacciones en Java.

Finalmente, analizamos algunas de las mejores prácticas al trabajar con transacciones en Java.

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
TUTORIAL DE JACKSON JSON (/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (/HTTPCLIENT-GUIDE)
DESCANSO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL DE PERSISTENCIA DE PRIMAVERA (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON SPRING (/SECURITY-SPRING)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJOS (/TAG/ACTIVE-JOB/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBE PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)