```java
/**
 *
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
 */
package org.apache.commons.dbcp.managed;

import org.apache.commons.dbcp.ConnectionFactory;

import javax.transaction.TransactionManager;
import javax.transaction.xa.XAException;
import javax.transaction.xa.XAResource;
import javax.transaction.xa.Xid;
import java.sql.Connection;
import java.sql.SQLException;

/**
 * An implementation of XAConnectionFactory which manages non-XA connections in XA
transactions.  A non-XA connection
 * commits and rolls back as part of the XA transaction, but is not recoverable since
the connection does not implement
 * the 2-phase protocol.
 *
 * @author Dain Sundstrom
 * @version $Revision$
 */
public class LocalXAConnectionFactory implements XAConnectionFactory {
    protected TransactionRegistry transactionRegistry;
    protected ConnectionFactory connectionFactory;

    /**
     * Creates an LocalXAConnectionFactory which uses the specified connection factory
to create database
     * connections.  The connections are enlisted into transactions using the specified
transaction manager.
     *
     * @param transactionManager the transaction manager in which connections will be
enlisted
     * @param connectionFactory  the connection factory from which connections will be
retrieved
     */
    public LocalXAConnectionFactory(TransactionManager transactionManager,
ConnectionFactory connectionFactory) {
        if (transactionManager == null) throw new
NullPointerException("transactionManager is null");
        if (connectionFactory == null) throw new
NullPointerException("connectionFactory is null");

        this.transactionRegistry = new TransactionRegistry(transactionManager);
        this.connectionFactory = connectionFactory;
    }
    public TransactionRegistry getTransactionRegistry() {
        return transactionRegistry;
    }

    public Connection createConnection() throws SQLException {
        // create a new connection
        Connection connection = connectionFactory.createConnection();

        // create a XAResource to manage the connection during XA transactions
        XAResource xaResource = new LocalXAResource(connection);

        // register the xa resource for the connection
        transactionRegistry.registerConnection(connection, xaResource);

        return connection;
    }

    /**
     * LocalXAResource is a fake XAResource for non-XA connections.  When a transaction
is started
     * the connection auto-commit is turned off.  When the connection is committed or
rolled back,
     * the commit or rollback method is called on the connection and then the original
auto-commit
     * value is restored.
     * </p>
     * The LocalXAResource also respects the connection read-only setting.  If the
connection is
     * read-only the commit method will not be called, and the prepare method returns
the XA_RDONLY.
     * </p>
     * It is assumed that the wrapper around a managed connection disables the
setAutoCommit(),
     * commit(), rollback() and setReadOnly() methods while a transaction is in
progress.
     */
    protected static class LocalXAResource implements XAResource {
        private final Connection connection;
        private Xid currentXid;
        private boolean originalAutoCommit;

        public LocalXAResource(Connection localTransaction) {
            this.connection = localTransaction;
        }

        /**
         * Gets the current xid of the transaction branch associated with this
XAResource.
         *
         * @return the current xid of the transaction branch associated with this
XAResource.
         */
        public synchronized Xid getXid() {
            return currentXid;
        }

        /**
         * Signals that a the connection has been enrolled in a transaction.  This
method saves off the
         * current auto commit flag, and then disables auto commit.  The original auto
commit setting is
         * restored when the transaction completes.
         *
         * @param xid  the id of the transaction branch for this connection
         * @param flag either XAResource.TMNOFLAGS or XAResource.TMRESUME
         * @throws XAException if the connection is already enlisted in another
transaction, or if auto-commit
         *                     could not be disabled
         */
        public synchronized void start(Xid xid, int flag) throws XAException {
            if (flag == XAResource.TMNOFLAGS) {
                // first time in this transaction

                // make sure we aren't already in another tx
                if (this.currentXid != null) {
                    throw new XAException("Already enlisted in another transaction with
xid " + xid);
                }

                // save off the current auto commit flag so it can be restored after
the transaction completes
                try {
                    originalAutoCommit = connection.getAutoCommit();
                } catch (SQLException ignored) {
                    // no big deal, just assume it was off
                    originalAutoCommit = true;
                }

                // update the auto commit flag
                try {
                    connection.setAutoCommit(false);
                } catch (SQLException e) {
                    throw (XAException) new XAException("Count not turn off auto commit
for a XA transaction").initCause(e);
                }

                this.currentXid = xid;
            } else if (flag == XAResource.TMRESUME) {
                if (xid != this.currentXid) {
                    throw new XAException("Attempting to resume in different
transaction: expected " + this.currentXid + ", but was " + xid);
                }
            } else {
                throw new XAException("Unknown start flag " + flag);
            }
        }

        /**
         * This method does nothing.
         *
         * @param xid  the id of the transaction branch for this connection
         * @param flag ignored
         * @throws XAException if the connection is already enlisted in another
transaction
         */
        public synchronized void end(Xid xid, int flag) throws XAException {
            if (xid == null) throw new NullPointerException("xid is null");
            if (!this.currentXid.equals(xid)) throw new XAException("Invalid Xid:
expected " + this.currentXid + ", but was " + xid);

            // This notification tells us that the application server is done using
this
            // connection for the time being.  The connection is still associated with
an
            // open transaction, so we must still wait for the commit or rollback
method
        }
        /**
         * This method does nothing since the LocalXAConnection does not support
two-phase-commit.  This method
         * will return XAResource.XA_RDONLY if the connection isReadOnly().  This
assumes that the physical
         * connection is wrapped with a proxy that prevents an application from
changing the read-only flag
         * while enrolled in a transaction.
         *
         * @param xid the id of the transaction branch for this connection
         * @return XAResource.XA_RDONLY if the connection.isReadOnly();
XAResource.XA_OK otherwise
         */
        public synchronized int prepare(Xid xid) {
            // if the connection is read-only, then the resource is read-only
            // NOTE: this assumes that the outer proxy throws an exception when
application code
            // attempts to set this in a transaction
            try {
                if (connection.isReadOnly()) {
                    // update the auto commit flag
                    connection.setAutoCommit(originalAutoCommit);

                    // tell the transaction manager we are read only
                    return XAResource.XA_RDONLY;
                }
            } catch (SQLException ignored) {
                // no big deal
            }

            // this is a local (one phase) only connection, so we can't prepare
            return XAResource.XA_OK;
        }

        /**
         * Commits the transaction and restores the original auto commit setting.
         *
         * @param xid  the id of the transaction branch for this connection
         * @param flag ignored
         * @throws XAException if connection.commit() throws a SQLException
         */
        public synchronized void commit(Xid xid, boolean flag) throws XAException {
            if (xid == null) throw new NullPointerException("xid is null");
            if (!this.currentXid.equals(xid)) throw new XAException("Invalid Xid:
expected " + this.currentXid + ", but was " + xid);

            try {
                // make sure the connection isn't already closed
                if (connection.isClosed()) {
                    throw new XAException("Conection is closed");
                }

                // A read only connection should not be committed
                if (!connection.isReadOnly()) {
                    connection.commit();
                }
            } catch (SQLException e) {
                throw (XAException) new XAException().initCause(e);
            } finally {
                try {
                    connection.setAutoCommit(originalAutoCommit);
                } catch (SQLException e) {
                }
                this.currentXid = null;
            }
        }

        /**
         * Rolls back the transaction and restores the original auto commit setting.
         *
         * @param xid the id of the transaction branch for this connection
         * @throws XAException if connection.rollback() throws a SQLException
         */
        public synchronized void rollback(Xid xid) throws XAException {
            if (xid == null) throw new NullPointerException("xid is null");
            if (!this.currentXid.equals(xid)) throw new XAException("Invalid Xid:
expected " + this.currentXid + ", but was " + xid);

            try {
                connection.rollback();
            } catch (SQLException e) {
                throw (XAException) new XAException().initCause(e);
            } finally {
                try {
                    connection.setAutoCommit(originalAutoCommit);
                } catch (SQLException e) {
                }
                this.currentXid = null;
            }
        }

        /**
         * Returns true if the specified XAResource == this XAResource.
         *
         * @param xaResource the XAResource to test
         * @return true if the specified XAResource == this XAResource; false otherwise
         */
        public boolean isSameRM(XAResource xaResource) {
            return this == xaResource;
        }

        /**
         * Clears the currently associated transaction if it is the specified xid.
         *
         * @param xid the id of the transaction to forget
         */
        public synchronized void forget(Xid xid) {
            if (xid != null && this.currentXid.equals(xid)) {
                this.currentXid = null;
            }
        }

        /**
         * Always returns a zero length Xid array.  The LocalXAConnectionFactory can
not support recovery, so no xids will ever be found.
         *
         * @param flag ignored since recovery is not supported
         * @return always a zero length Xid array.
         */
        public Xid[] recover(int flag) {
            return new Xid[0];
        }

        /**
         * Always returns 0 since we have no way to set a transaction timeout on a JDBC
connection.
         *
         * @return always 0
         */
        public int getTransactionTimeout() {
            return 0;
        }

        /**
         * Always returns false since we have no way to set a transaction timeout on a
JDBC connection.
         *
         * @param transactionTimeout ignored since we have no way to set a transaction
timeout on a JDBC connection
         * @return always false
         */
        public boolean setTransactionTimeout(int transactionTimeout) {
            return false;
        }
    }

}
```